DOCUMENT RESUME

ED 168 570 IR 007 111

AUTHOR Atwood, Michael E.; And Others
TITLE Planning as a Process of Synthesis. Technical Report.
INSTITUTION Science Applications, Englewood, Colo.
SPONS AGENCY Office of Naval Research, Washington, D.C. Personnel and Training Branch.
REPORT NO SAI-78-144-DEN
PUB DATE Dec 78
CONTRACT N-00014-78-C-0165
NOTE 107p.

EDRS PRICE MF01/PC05 Plus Postage.
DESCRIPTORS Behavioral Science Research; Cognitive Objectives; *Cognitive Processes; Memory; Models; *Planning; *Problem Solving; *Psychological Studies; Schematic Studies
IDENTIFIERS *Software Design

ABSTRACT

The theoretical framework for investigating human planning behavior and the procedures, results, and implications of two experiments involving planning in the domain of software design are presented in this report. A plan is defined as a hierarchical structure that underlies the solution to a problem; planning is the process of constructing this structure. This framework makes various assumptions about the nature of a plan and the construction of the planning structure. Experiment one involved the collection of long, thinking-out-loud protocols from three highly experienced computer scientists. Experiment two involved ten undergraduate college students enrolled in an "assembly language" course who were instructed to solve a design problem similar to the one given to the experts in either a written-solutions condition or a written-and-think-aloud condition. Examination of the experimental behaviors indicate that completed plans can be characterized in a variety of ways. Expert subjects differ from less experienced subjects in the knowledge structures that can be retrieved from long-term memory and incorporated into a plan. (Author/CWM)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED 168570

U S DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
NATIONAL INSTITUTE OF
EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

PLANNING AS A PROCESS OF SYNTHESIS

Technical Report
SAI-78-144-DEN

December 1978

Michael E. Atwood
Science Applications, Inc.

Peter G. Polson and Robin Jeffries
University of Colorado

H. Rudy Ramsey
Science Applications, Inc.

Reproduction in whole or in part is permitted for any purpose of the United States Government.

This research was sponsored by the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval Research, under Contract No. N00014-78-C-0165, Contract Authority Identification Number, NR157-414.

IR 007 111

Approved for public release; distribution unlimited.

Science Applications, Inc.
40 Denver Technological Center West, 7935 East Prentice Avenue, Englewood, Colorado 80111, 303/773-6900
Other SAI Offices: Albuquerque, Ann Arbor, Arlington, Atlanta, Boston, Chicago, Huntsville, La Jolla, Los Angeles, McLean, Palo Alto, Santa Barbara, Sunnyvale, and Tucson.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Planning as a Process of Synthesis		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER SAI-78-144-DEN
7. AUTHOR(s) Michael E. Atwood, Peter G. Polson, Robin Jeffries, & H. Rudy Ramsey		8. CONTRACT OR GRANT NUMBER(s) N00014-78-C-0165
9. PERFORMING ORGANIZATION NAME AND ADDRESS Science Applications, Inc. 7935 E. Prentice Avenue Englewood, CO 80111		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR157-414
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel & Training Research Programs Office of Naval Research Arlington, VA 22217		12. REPORT DATE December 1978 13. NUMBER OF PAGES 106
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Planning
Problem Solving
Cognitive Psychology

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes a theoretical framework for investigating human planning behavior and presents the results of two experiments in the domain of software design. A plan is defined as a hierarchical structure that underlies the solution to a problem; planning is the process of constructing this structure. This framework assumes that: (a) a plan is a series of abstractions of the final solution, ranging from schematic, high-level plans to detailed plans that are actually transformed into a solution to the problem; (continued 2nd page)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

#20 (Continued): (b) a plan is constructed by a process similar to stepwise refinement; (c) planning involves the utilization of previously learned schemata; (d) various components of the plan, or even the entire plan, can be retrieved from long-term memory and incorporated into a solution to the problem; and (e) planning involves the synthesis of many types of knowledge structures.

Our experimental results indicate that completed plans can be characterized as procedural nets, but that plan structures can be constructed in a variety of ways. Further, expert subjects differ from less experienced subjects in the knowledge structures that can be retrieved from long-term memory and incorporated into a plan. The implications of these results on our theoretical framework and for future research are discussed.

ABSTRACT

This report describes a theoretical framework for investigating human planning behavior and presents the results of two experiments in the domain of software design. A plan is defined as a hierarchical structure that underlies the solution to a problem; planning is the process of constructing this structure. This framework assumes that: (a) a plan is a series of abstractions of the final solution, ranging from schematic, high-level plans to detailed plans that are actually transformed into a solution to the problem; (b) a plan is constructed by a process similar to stepwise refinement; (c) planning involves the utilization of previously learned schemata; (d) various components of the plan, or even the entire plan, can be retrieved from long-term memory and incorporated into a solution to the problem; and (e) planning involves the synthesis of many types of knowledge strtuctures.

Our experimetal results indicate that completed plans can be characterized as procedural nets, but that plan structures can be constructed in a variety of ways. Further, expert subjects differ from less experienced subjects in the knowledge structures that can be retrieved from long-term memory and incorporated into a plan. The implications of these results on our theoretical framework and for future research are discussed.

Table of Contents

Introduction	1
Task Description	6
Theoretical Framework	10
Research on Planning: A brief Review	20
Method for Data Analysis	37
An Investigation of Planning Behavior	39
Experiment 1. The Problem Solving Behavior of Experts	39
Experiment 2. Alternative Methods of Protocol Collection	67
Conclusions	77
Appendix A — Software Design Practices	83
References	98

This report presents our initial attempts to formulate a theoretical framework that characterizes planning in complex task environments. We also report the results of two experiments using the task of software design. The focus of this research is on how subjects solve complex problems in domains that require the application of a large amount of background knowledge. Such problems have been characterized by Bhaskar and Simon (1977) as "semantically rich domains."

If an individual is going to solve a complicated problem employing a digital computer, that individual has to write a computer program. If the program is of any complexity a person who simply starts writing code without any preparation has little chance of generating a successful solution, i.e., a working program. A solution to the problem must begin with the specification of a design for the program -- a plan for the ultimate solution. Thus, we are going to use the task of software design to study planning in complex, semantically rich domains.

Computer programming has another very interesting characteristic, in that it is a general problem solving skill. In order to write a program to solve a specified problem, it is required that one integrate several quite different kinds of knowledge. Examples are expertise in programming and one's understanding of the problem to be solved.

Minsky (1975) has argued that the solution to almost any kind of challenging problem involves the integration of different viewpoints or different kinds of knowledge. Minsky presents the following example involving an automobile.

> "Sometimes in 'problem solving' we use two or more descriptions in a more complex way to construct an analogy or

> to apply two radically different kinds of analyses to the same situation. For hard problems, one 'problem space' is usually not enough.
>
> "Suppose your car battery runs down. You believe that there is an electrical short and blame the generator.
>
> "The generator can be represented as a mechanical system; the rotor has a pulley wheel driven by a belt from the engine. Is the belt tight enough? Is it even there? The output, seen mechanically, is a cable to the battery or whatever. Is it intact? Are the bolts tight? Are the brushes pressing the commutator?
>
> "Seen electrically, the generator is described differently. The rotor is seen as a flux linking coil, rather than as a rotating device. The brushes and commutator are seen as electrical switches. The output is current along a pair of conductors leading from the brushes through control circuits to the battery.
>
> "Thus, we represent the situation in two different frame systems." (Minsky, 1975, pg. 256).

The hard problems that Minsky refers to are, in effect, problems whose solutions require two or more different perspectives and the integration of two or more knowledge structures. In Minsky's example, either representation may be adequate, buc it is more likely that a successful solution will require simultaneous consideration, or integration, of these representations. In addition, another type of knowledge would be required. Minsky's representations are sufficient to understand the problem; they are not sufficient, however, to actually implement a solution. For example, the successful problem solver would need to know how to tighten a belt, how to measure current flow, etc. Integration of this type of knowledge into the ultimate solution is clearly required.

When we attempt to write a computer program we are in a very similar situation. Knowledge of programming, per se, provides us with a set of tools. These tools in themselves are not adequate. We have to understand what they are going to be used for. Thus, we must understand the problem to be solved using the computer; this is a

second type of knowledge. In summary, we are investigating planning behavior in problems whose solution requires the integration of multiple knowledge domains.

The remainder of this paper is organized into several sections. We will begin with a discussion of the concept of planning. We then describe the task domain of software design and our rationale for its use in this research. Following this, we present a detailed discussion of our theoretical framework that motivates the research reported in this paper. Next, we present a brief review of previous research on planning and problem solving with particular emphasis on tasks requiring a large amount of background knowledge, that is, semantically rich domains, and on issues involving expertise. Finally, we present our experimental results and make conclusions derived therefrom.

In an appendix to this paper, we present a brief summary of the literature on software design. This section is intended to introduce the reader to the concepts and terms involved in this area. An interesting aspect of the task of software design, like chess and many other real world tasks, is that this task has its own literature and we will find that experts in this area can assist us in our psychological analysis of performance on software design tasks.

PLANNING -- A PRELIMINARY DEFINITION

This section presents our initial attempt to define the concept of a plan and to distinguish this concept from the process of planning. Our definition is not atheoretical, nor do we feel that an atheoretical definition of plan is possible. Thus, the definition presented in this section anticipates many of the theoretical arguments that we will

present later.

Following Miller, Galanter, and Pribram (1960) we will define a plan to be a hierarchical structure that underlies the sequence of operations necessary to solve a problem. We define planning to be the generation or specification of this hierarchical structure. Our definition is obviously extremely general, and is consistent with notions like "deep structure," "case frame," "frame," "script," and a number of similar concepts that are very popular in cognitive psychology today.

In order to further elucidate our notions of "plan" and "planning", we would like to construct several illustrations. Imagine a series of individuals arranged along a continuum according to their software design skills, ranging from expert to novice. The basic assumption in the literature today is that very different kinds of cognitive structures underlie expert versus novice problem solving behavior.

Let us consider the solution of some fairly straightforward problem. If the expert has had a large amount of experience with this specific type of problem, then the expert may retrieve its solution and simply present it to us. One might almost be tempted to say that we were looking at a retrieval process rather than a problem solving process. On the other end of the continuum, the novice may have some vague understanding of the knowns and unknowns involved in the problem, but no knowledge of the structure of the sequence of the operations that would actually solve this problem. Therefore, the novice is reduced to using some variation of trial-and-error search. Individuals that are in the intermediate range of the continuum will actually

construct a solution to their problem. Although they don't have a schema or plan memorized, their knowledge of the task domain is sophisticated enough that they are able to generate a solution plan and, finally, a sequence of operations that will successfully solve the given problem.

We assert that the expert retrieves a plan from long-term memory and then proceeds to execute this plan. Since such plans can be represented as schemata, we would characterize this behavior as being schema-driven. On the other end of the continuum, the behavior of the novice essentially involves trial and error search. There are no underlying structures that provide guidance to this problem solving behavior, and, as a result, this person is very unlikely to find a solution to the problem. For any interesting problem, the search space is simply too large. The individual with an intermediate amount of knowledge about the task is not reduced to trial-and-error search. This individual, however, is not able to retrieve an already constructed plan. Thus, we will characterize this individual as indulging in the activity of planning; he must construct a solution plan.

In summary, we have characterized a plan as a hierarchical structure that represents a sequence of actions and the process of planning as the process of generating this hierarchical structure. In the task domain to be used here, a computer program represents a solution to the problem. A software design for this program is the plan that underlies the solution. The process of actually constructing a given design is a planning activity.

TASK DESCRIPTION

The experimental task used in the experiments to be reported in this paper is software design. Software design is the process of translating functional specifications into a structural description of a computer system that will satisfy these specifications. There are, in general, three components of this structural description. First, the description takes the form of a "modular decomposition". That is, the original functional specifications are decomposed into a collection of modules, or substructures, each of which satisfies only part of the original specifications. Second, these modules must communicate in some way, and the designer must specify the interrelationships and interactions of these modules. Third, design may include a definition of the data structures that are required to satisfy the functional requirements.

It is convenient to think of the functional specifications as specifying the properties that are desired. The design identifies the functions that can satisfy these properties. In actual practice, a design specifies what must be done in order to meet the functional specifications. How these functions are to be accomplished is left to the programmer.

In order to provide an example of a software design, consider the following problem described by Wasserman (1977). The functional specifications could be presented to the designer as follows:

"A medical center needs a master patient index. The master patient index file will be kept on disc for fast on-line access. The purpose of this index is to identify a patient. The following specifications must be met.

1. Input and output will be performed at various locations. Each location, and each person at a location, should have different access privileges.

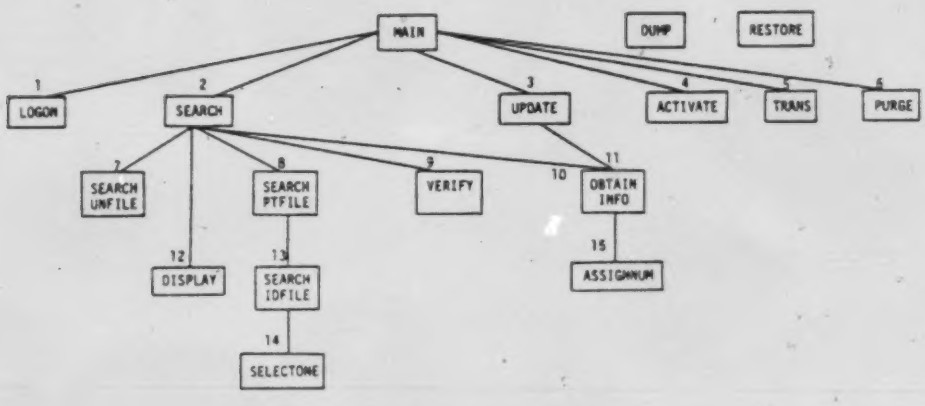
2. Each patient will be identified by a "patient unit number"; if a number has not previously been assigned, one must be assigned.

3. Each patient's identification will be confirmed by asking questions about his personal background (e.g., mother's maiden name).

4. A mechanism must be provided for purging, to off-line storage, patient files that are infrequently accessed and for retrieving, from off-line storage, files for reactivated patients.

5. A daily record of all transactions should be written on magnetic tape."

Notice that this description specified the functions that the system must satisfy, but does not specify the form of the design. A potential design for such a sytem (adapted from Wasserman) is represented as:



Each of the rectangles names a module of the completed system. The lines connecting modules indicate interactions between modules. A designer would also, given the above form of design, provide a description of each module and assumed file structure. For example:

UNFILE: file of patient unit numbers

SEARCH: Search the data base for patient identification; obtain identification if patient is not found in data base; update data base appropriately

VERIFY: request verification of patient identification information

LOGON: determine user authorization to access data base etc.

Design, in its broadest sense, appears to be an excellent vehicle for the study of planning in any kind of problem solving domain. Design, whether it is concerned with construction of a building, development of a complicated computer program, or a plan of action to control a large industrial enterprise, is essentially a plan describing the goals that are necessary to achieve the ultimate objective of the plan and the operations necessary to achieve those goals. The design is the plan or structure underlying a given computer program or software system. In large, or even moderately large software systems, software design is considered to be a separate task from other programming activities. Designers produce high level descriptions of the system to be implemented, and then these descriptions are given to other individuals who will perform the actual implementation, i.e., programmers. Recall that earlier we characterized a plan as a hierarchical structure underlying a sequence of operations or actions that solve a given problem. The computer program is that sequence of

actions; the design is the plan.

We have selected software design because we feel that it maximizes our chances of understanding planning behavior. Another reason is that programmers and systems designers have a common terminology for describing elements of their solutions -- the technical vocabulary of computer science. This common vocabulary simplifies data collection and analysis and makes a much more rigorous interpretation of our protocols possible. Both the experimenters, who are software designers of various levels of expertise, and the subjects have a common and reasonably well defined vocabulary for communicating the various elements of this underlying hierarchichal structure. Third, it is possible to present a wide range of problems in the context of the task domain of software design. Finally, the task environment of software design appears to be very compatible with the theoretical framework that we are using. In addition, there is a large literature on software design, and this literature provides additional insights into the structure of this task and the necessary background information required to construct a theory of human performance in this domain.

THEORETICAL FRAMEWORK

In this section we present the theoretical framework underlying our research on planning and software design. Our thinking has been very strongly influenced by two different literatures. The first is the work on schema-based representations of knowledge. The second is the work on planning that has been done in connection with numerous robot projects. In addition, our framework is consistent with, although not directly influenced by, some of the concepts derived from formal literature on software design that will be reviewed in Appendix A.

We begin this section by reiterating our definitions of plan and planning. We then go on to outline our theoretical framework. Next we present a fairly detailed discussion of a model of planning developed by Sacerdoti (1975). We then present a more detailed articulation of our theoretical ideas incorporating recent thinking on the representation and utilization of knowledge. In the final section, we will also briefly consider other theoretical work, although derived from different sources, that has similarities to our own.

To recapitulate our earlier definitions, we defined a plan to be a hierarchical structure that underlies any sequence of actions or operations. Planning is the process of generating this hierarchical structure. However, it is necessary that we distinguish between plan generation and plan retrieval.

In work that we will review later, there is a growing consensus that experts in a given domain have a large number of very specific plans for the solution of problems in the domain in which they are expert. In a situation where an individual is essentially executing a

pre-stored plan, we use the term plan exeuction. Although it may not appear so at first glance, plan retrieval is a form of planning. The plans being utilized by an expert are often extremely schematic and their adaptation to a specific situation is a nontrivial process. Still the solution plan is not being generated "from scratch." The planning process is greatly simplified due to the individual's experience with similar tasks. Plan generation refers to a situation where a novice or an expert is attempting to generate a new plan. In the studies that we will report later in this paper, none of our software designers were expert in the particular kinds of problems that they were given in our experiments. Thus they were faced, in part, with the task of generating a new plan.

In summary, a plan is a hierarchical structure that underlies the solution to a problem. Planning can be divided into two classes of processes. Plan retrieval is the process of making specific a schematic solution to a given task or class of problems. Plan generation is the process of generating a new plan for the solution to a problem. The theoretical framework to be outlined below makes the basic assumption that in a software design task the development of a plan for a novel problem is, for the expert, in fact, a mixture of plan generation and plan retrieval.

Software design can be considered a "problem of arrangement." As Greeno (1973) points out, the archetype of a problem of arrangement is the anagram. In software design we think that the elements that are being arranged are the subject's knowledge of specific kinds of operations, such as input-output processes, sorting, management of linked lists, and other such fairly specific concepts in the domain of

computer science. It is the proper arrangement of these known elements, or processes, that a compucer system can execute, that defines the eventual solution to a given programming problem, and the task of the designer is to arrange those elements in the proper order. The framework outlined below incorporates five basic assumptions about planning behavior in complex tasks, in particular the task of software design. We begin by summarizing these main points.

First, we propose that a complete plan is a series of abstractions of the final solution to the problem ranging from very schematic or high level plans to detailed plans that can actually be transformed into a solution to the problem. Second, a plan is generated by a process very similar to stepwise refinement. That is, plans are generated primarily in a top-down and breadth-first manner, with each level being more detailed than its predecessor. Third, planning involves the utilization of previously learned schemata. These schemata may be very general, such as the "input-process-output" schema from computer science, or may include more specific schemata, such as the subject's knowledge of how to manipulate elements that are represented by a linked list. Fourth, various components of a total plan, or even complete solution plans, can be retrieved from long term memory and incorporated into the solution of a specific problem by the process of plan retrieval. Finally, we assume that planning, in complex tasks, involves the synthesis of many types of knowledge structures.

Our basic ideas about the planning process were stimulated by Sacerdoti's (1975) description of NOAH. NOAH (Nets of Action Hierarchies) is an integrated problem solving system that utilizes

stored information about the task domain to generate a complete plan for the solution to a given problem. The system solves the problem by creating a hierarchical structure that represents a solution plan for a given problem at greater and greater levels of detail. The top node or level in the hierarchy is a one step solution to the problem, essentially a statement of the goal "solve the problem." The successors of this top node are the major subgoals of the completed plan. Successive levels of this hierarchy are more and more detailed plans. The bottom level nodes are a solution to the task in terms the primitive actions of the task domain. The nodes at any given level are linked together by predecessor and successor relations that define a partially ordered sequence of operations. Sacerdoti calls this completed structure "a procedural net". In the discussion that follows we will use the terms "procedural net" and "plan" interchangeably.

NOAH uses an iterative procedure in generating a plan to solve a problem. Each cycle of the iteration begins with the expansion of each node into its successors. These successors are a complete subplan for achieving the goal defined by the parent. However, there is no guarantee that the sequence of individual subplans is a correct solution. The completion of one subplan in the sequence may make it impossible to achieve a necessary step in a later subplan in the series. A correct plan is generated from this sequence of subplans by a collection of "critics". These critics make the plan consistent by reordering the steps and eliminating redundant ones. In addition, the critics attempt to deal with conflicting preconditions defined by various subplans. The cycle then repeats with a new plan being synthesized at the next greater level of detail. This iterative

process generates a procedural net for the solution to the problem.

The process we have been describing is very closely related to the notion of stepwise refinement and similar concepts that have appeared in the software design literature. In the empirical work to be reported later, the primary focus of our current research is the dynamics of the planning process. We will argue that human problem solvers develop plans in a manner that is similar to NOAH.

The basic assumption underlying NOAH is that one can create a rough plan for solving a problem by ignoring much of the detailed information contained in the original statement of the problem. This rough, or abstract, plan is then refined into a more detailed, complete plan to solve the problem. The notion of developing an initial solution plan using a simplified version of the problem was first proposed by Newell, Shaw, and Simon (1963). This original proposal of planning by abstraction is described by Newell and Simon (1972) in some detail. This type of planning process has been very influential in work that was published during the last few years in the robotics literature and is most apparent in ABSTRIPS (Sacerdoti 1974), (cf. Banerji and Ernst, 1977). Sacerdoti (1975) assumed that each node describing a subgoal at a given level of abstraction contains all the information necessary to construct a solution plan at the next level of detail. We will not make use, however, of the particular representation schemata that Sacerdoti incorporated into his system. We will focus on the dynamics of the solution process as described by Sacerdoti and turn to the literature on schema-based representations of knowledge to deal with how the knowledge incorporated into the procedural net is actually represented.

For different kinds of problems and subjects (e.g., novices and experts), the possible representations of plans can vary both in the numbers of levels of abstraction as well as the generality of the plan at any given level of abstraction. We propose that we can characterize the subject's problem space at any given point in time as an incomplete procedural net. We assume that the top node of the net corresponds to the subject's representation of the goal to be achieved, that goal being to solve the problem. The lowest level of the subject's representation is the sequence of operations used to actually solve the problem within the task domain. The intermediate level represents the subject's understanding of both abstract plans and the particular subgoal sequences that define the major steps in the process of solving the problem.

It is important to realize that a subject's procedural net may be incomplete and/or incorrect. In the extreme case of simple transformation problems, the subjects' representation consists of the top node (i.e., the goal stating "solve the problem"), and the two end points of the move sequence, the initial situation and the goal state of the problem. The lack of planning in such problems reflects the fact that the subjects' understanding of the problem is very incomplete. Their knowledge of the task does not permit them to generate the hierarchical structure that underlies the sequence of operations that will eventually solve the problem.

We will partition the various levels of a procedural net into three types of plans — abstract plans, detailed plans, and the operation or move sequence level. The top levels of a procedural net represent an abstract plan for solving the problem. These abstractions

eliminate many of the details of the original problem in an attempt to identify the major components or subproblems of the task as originally stated. This identification, however, can be rather crude, and this level of plan is not necessarily an accurate description of an achieveable solution. For example, Sacerdoti points out that an abstract plan may simply identify the major subproblems to be solved in the process of achieving the stated solution. This initial abstraction can ignore the very critical issue of the order in which the subproblems must be attacked and solved.

The detailed plans are represented by the middle levels of the net. These structures define the details of a solution to a problem. It is this level that we feel most closely corresponds to the structures generated by NOAH. In addition, we will argue later that detailed plans are generated from information stored in long term memory, and that they characterize the operations that will solve the problem. The bottom level of a procedural net is the problem graph defined by the primitive actions or operations of the problem. In the domain of computer programming, the abstract plan and the detailed plan are the software design. The complete design is turned over to a programmer who generates the program, or the operation sequence level of the net, and creates an operational or running program that solves the given problem.

NOAH, like a large number of other formalisms characterizing complex problem solving, assumes that a problem is solved by the process of problem reduction. The problem as given is reduced by a decomposition process into a manageable series of subproblems. The "critics" in NOAH deal with interactions between solutions to

subproblems. The major difference between our theoretical thinking and Sacerdoti's NOAH has to do with how we wish to represent the knowledge that is incorporated into the completed plan.

We assume that the knowledge underlying the solution to a problem, the knowledge incorporated into the procedural net, can be partitioned into two categories. The first is information that a subject must have in order to understand the description of the problem that is to be solved -- the information necessary to understand the purpose of the computer system that is to be designed. This knowledge would include the knowledge of topics like physics, chemistry, accounting, statistics, or any substantive discipline that could utilize computer programs to accomplish a task. The second category includes the problem solver's knowledge of design techniques, general knowledge of computer science, knowledge of specific processes like sorting or memory management, and knowledge of the details of a specific computer system. Our primary assumption is that the knowledge incorporated into the procedural net which describes the design for a given system is a synthesis of these two knowledge domains. An individual's knowledge of the problem area is the basis for the understanding of the problem statement and the identification of the major components of the problem. This knowledge enables an individual to parse, or define, the principal subproblems that must be solved in order to achieve the solution of the given problem.

We will assume that an individual's knowledge of software design and computer science can be roughly partitioned into three categories. The first includes general knowledge about the overall process of design and the particular kinds of resource allocation heuristics that

an individual uses to guide design behavior. Expert designers are well aware of different kinds of design styles. Furthermore there are probably significant, systematic differences in the kinds of resource allocation decisions designers make. For example, "how detailed should I make this design?", "should I optimize speed or storage requirements?", etc.

The second category is a very generalized schema which frequently takes the form of "input-process-output." For the overall process, or any subprocess that is going to be part of a design, the designer must specify the inputs to that routine or process, the process or the manipulations to actually be performed by the computer, and then the format of the results or the output. This schema itself is not the solution to any programming problem; it is simply too general. This schema is an agenda that must be completed before the design for a given component of the problem can be considered finished.

The third category of knowledge the software designer has is a large collection of quite specific pieces of information involving various kinds of techniques in computer science and the specific computer system involved. An expert designer has at his command a large amount of knowledge about specific techniques for sorting, pattern matching, list management, storage management, etc. These techniques are often stored in memory in the form of quite general structures that then are adapted to a given problem environment. Knowledge of the structure of the particular programming system available to the designer, such as the operating system or the programming language that the design is to be implemented in, is represented similarly. This type of knowledge identifies any

constraints on the design that may be imposed by the environment that the solution will actually be implemented in.

We assume that the top levels of the procedural net are generated by a direct combination of an individual's knowledge of the problem to be solved with the very general templates or schemata specified by the particular implementation domain (e.g., software design). In our case, recall that this schema is "input-process-output." This synthesis results in a top level abstract plan that identifies the major functional elements of the eventual design. The rest of the plan consists of descendants of these initial nodes.

Once the major elements or subgoals of the abstract plan have been identified, the expansion of these subgoals into detailed plans is in large part controlled by the subject's knowledge of computer techniques and design. At this point, the subject has identified what must be done in order to solve the given problem and begins to focus on how these subgoals can be accomplished. Once a given abstract subgoal has been expanded into a detailed collection of subgoals, the expansion of these detailed goals involves very specific schemata that describe how the particular set of operations used to accomplish this subgoal are in fact accomplished on a real computing system. Thus the construction of the detailed plan essentially involves synthesizing and arranging of elements that are retrieved from long term memory. These are elements of the subject's knowledge of very general classes of operations that can be performed on digital computers, e.g., sorting, and the subject's detailed knowledge of the particular computer system.

RESEARCH ON PLANNING: A BRIEF REVIEW

In this section, we are going to discuss a large number of topics that are directly or indirectly relevant to the theoretical framework that we outlined earlier. We are going to focus on three related literatures. The first is the work in the area of artificial intelligence that is closely related to NOAH. The second is the psychological literature on planning. The third is the literature on schema-based representations of knowledge. Although there is not a large literature in psychology on planning, per se, it is clear that given our very general definition of planning, that "plans" and "planning" are very closely related to many topics that are central to current interests in cognitive psychology. Finally, we will consider the relations between problem understanding and problem solving.

RELATED WORK

Two projects have recently been reported in the artificial intelligence literature that make assumptions about planning and problem solving that are very similar to those that underlie both our theoretical framework and NOAH. The first is a system that writes computer programs -- Program Writer (Long, 1977). The second is a problem solving system developed by Sussman (1977) -- PSBDARP (Problem Solving By Debugging Almost Right Plans).

The Program Writer is an artificial intelligence program that accepts high level specifications for a computer program in a limited domain (withdrawals and deposits to bank accounts) and generates appropriate algorithms and data structures to implement the program, or to solve the problem. The central thesis of Long's (1977) work is that

writing a program requires several different types of information and that the process of generating such a program involves stepwise refinement, or iterating the solution through several levels of abstraction.

Long assumes that the knowledge necessary to successfully generate a program is stored in a collection of substructures that he refers to as "models." Five models are involved in the current version of Program Writer. The first is the domain model, which contains information about the application area (in this case, banking transactions). For example, the domain model specifies such things as that making a deposit implies that a person has money to be maintained at the bank, that the person owns the money, etc. The remaining four models are primarily concerned with the knowledge required to produce functioning programs. The four models involved are: (1) the argument passing and control model, (2) the data model, (3) the input/output model, and (4) the target language model. The information contained in these substructures is used to guide the program generation process, which is under overall control of a design model. The design model essentially uses the process of stepwise refinement to generate a running program. Thus, Long's design model is his theory of programming and problem solving.

In general, the assumptions underlying Long's Program Writer are very similar to the ideas underlying Sacerdoti's NOAH system and our theoretical framework. All generate plans in a hierarchical, top-down manner. Further, all develop a rough plan for solving a problem by ignoring much of the detailed information in the original statement of the task and then expanding this plan in more detail.

Both Sacerdoti and Long also incorporate some mechanism to assure that the global constraints of the problem (essentially ignored in the high level plans) are satisfied; in Sacerdoti's work these are the critics and in Long's these are the models. Sacerdoti and Long make very different assumptions about how knowledge is organized. Recall that Sacerdoti assumed that the knowledge necessary to generate the next level of the plan was contained in individual nodes associated with each higher level goal. Long assumes, as we do, that knowledge is organized into complex, schema-like structures -- the models.

Sussman (1977) has attempted to model the processes involved in solving problems in electrical circuit design. Sussman's research is concerned not only with how plans, or designs, are created, but also with how incorrect plans are modified and, to a degree, with what types of knowledge are involved in plan construction and modification. PSBDARP (Problem Solving By Debugging Almost-Right Plans) is a system that incorporates two stages that can be employed either independently or in succession.

In the first stage, attempts are made to retrieve previously generated solutions to problems that are similar or identical to the current problem. If a solution to a similar problem is found, its applicability to the current problem is evaluated. If it is judged appropriate, the previous solution is applied; if it is not appropriate, an attempt is made to isolate and correct the discrepancies ("bugs") that prevent successful application. Thus, this stage involves the retrieval of existing solutions or solution schemata and, if required, making modifications to these schemata.

The second stage is entered only if no applicable solution schema

is retrieved. In this stage, an attempt is made to decompose the given problem into subproblems and, if this fails, to find an alternative representation of the problem. In either case, the intent is to produce a collection of subproblems that appear consistent with existing solution schemata. Any bugs that are detected in the schemata are resolved by recursive calls to PSBDARB.

In addition to previously generated, or known, solutions, Sussman assumes that there are knowledge structures that also exist to describe rules for making problem representaton changes and problem decompositions. Like the solution schemata, this information is stored or organized with respect to its applicability to certain types of problems. These structures are the problem solving processes that are incorporated into PSBDARP.

Long's (1977), Sussman's (1977), and our work bear striking similarities. Long (1977) assumes, as we do, that planning and problem solving utilize a series of representations of the problem that involve the specification of increasing levels of detail -- the process of stepwise refinement. We share with Sussman the view that problem solving in complex domains can be characterized as a problem of arrangement. Many of the processes incorporated in Sussman's theory are intended to tailor or modify previous knowledge so that it fits the constraints of the current problem; that is, the emphasis is on debugging.

While Long and Sussman are taking a fairly strict artificial intelligence approach, Levin (1976) has attempted to develop a theory of the software design process that is consistent with current thinking in cognitive psychology. Levin assumes that design can be viewed as

involving three fundamental processes -- "selecting problems to work on, gathering needed information for the solution, and generating solutions", (Levin, 1976, p. 2). Levin focuses on the problem selection processes. He makes a distinction between global information (strategies) and local information (constraints) and makes three assertions concerning the problem selection process:

> "1. Local constraints play an important role in problem selection and account for a significant percentage of new problem selections during design. (A local constraint is one that has been introduced only within the scope of the most recently selected problem).
> 2. As strategy and constraint information ages in working storage, the probability that it will be used as a problem source decreases.
> 3. The required presence of strategies in working storage and prior use of local constraints limits the use of strategies as a problem source." (pp. 10-11).

Levin is dealing with the information that controls the process of stepwise refinement as the design iterates to greater and greater levels of detail. In effect, he is attempting to provide a description of how the information used in Long's (1977) models is utilized in the process of solving the problem. He has decided to concentrate on the resource limitations, in particular memory limitations, that dictate much, if not all, of human problem solving performance.

Levin then proceeds to develop a simulation model based on these assumptions. This model takes as input the protocol of an experienced designer working on a fairly complex problem and produces as output a list of subgoals generated by the designer. It is not immediately clear, however, whether this model accurately describes human behavior. As Levin notes, the model is not able to evaluate decisions or constraints or to determine the relative importance of one subgoal over another. It processes the input protocols, but does not "understand"

the design task, and the concept of a goal is completely missing.

As used by Levin, the term "strategy" refers to "plans for achieving the solution to a problem. A strategy describes a sequence of activities (subproblems), which when worked on may achieve a problem solution" (p. 9). Levin further distinguishes between "local" strategies and "global" strategies in a manner which corresponds closely to NOAH's detailed and abstract plans. Further, these strategies can be characterized as previously learned schemata that the designer brings to bear on the design task. Although Levin's concepts are not entirely worked out, his model seems to be generally consistent with the other research discussed in this paper.

PSYCHOLOGICAL RESEARCH ON PLANNING

There is a limited research literature in psychology that directly relates to planning, but a very large literature that bears directly or indirectly on the issues discussed in this paper. The first is the literature in the early and mid 1970's on problem solving processes in transformation problems. The second is the characterization of the problem solving behavior of experts in various problem domains. The third topic is Newell and Simon's (e.g., 1972) work on planning by abstraction which is a basic concept that is incorporated into NOAH, and which was discussed earlier.

Greeno (1974) and his students (Thomas, 1974; Egan and Greeno, 1974) argue that simple forward planning processes could account for the performance of subjects on various transformation problems. Recall that transformation problems are tasks in which the subject has a definite start state and goal state, and the solution to the problem

involves the application of a limited number of well-specified operations that transform the start state into intermediate states and, finally, into the goal state. Examples of transformation problems are water jug problems, river crossing problems, and the Tower of Hanoi. These simple, puzzle-like tasks have been a major focus of modern work in problem solving. Greeno and his students have argued that the problem solving behavior of subjects working on river crossing tasks (Thomas, 1974), and the Tower of Hanoi (Egan, 1973) can be explained by various kinds of simple forward planning mechanisms. Atwood and Polson (1976) and Jeffries, Polson, Razran, and Atwood (1977) have challenged these conclusions. They developed three-stage quantitative models of the move selection process that assume that subjects use only local information in selecting the next move. They were able to account, in particular for the water jug task, for all aspects of subjects' behavior that Greeno and his colleagues had argued required assuming some kind of planning process. Polson and his colleagues concluded that since they could provide quantitative explanations for subjects' performance in this class of task by assuming no planning processes, the assumptions of Greeno and his colleagues must have been incorrect.

The "no planning" finding of Polson and his colleagues is easily explained in the context of the theoretical paradigm presented in this paper. The elementary puzzles that make up many of the transformation problems that are frequently used in empirical studies are very difficult for naive subjects. Although subjects understand the characterizations of the goal and start states and the legal moves that can be used to solve the problem, they have no knowledge of the structure of the move sequence that will ultimately solve the problem.

In the terminology of the framework presented in this paper, these subjects are unable to generate abstract and detail level plans that underly the solution sequence. Thus, they are reduced to solving problems of this class by some type of means-ends driven trial-and-error search. It is interesting to note that all successful quantitative models of this class of tasks make similar "no planning" assumptions (Simon and Reed, 1976; Atwood and Polson, 1976; Jeffries, et al, 1977). In conclusion, elementary transformation problems are difficult because a simple description of the rules of the problem seems to give subjects little or no information about the structure of the sequence of moves that will solve the task. This reduces the subject to proceeding by some sort of sophisticated trial-and-error search that is guided only by information about the next move.

Although several investigators have incorporated some type of no planning assumption, none have argued that this is a general conclusion. In fact, it is obvious that there must be some kinds of planning behavior underlying successful solution of any really challenging problem. The simple counterargument to straightforward trial-and-error search schemes is that as the structure of the problem becomes more complex, the search space grows explosively, even exponentially, and trial-and-error search will not converge on a solution in any reasonable period of time. Identical problems with the explosive growth of the search space have forced designers of robot systems to develop sophisticated planning processes so that computers can solve problems in such a seemingly simple domain as maneuvering in a three-dimensional world.

In the last few years, research on problem solving has been

concerned with tasks that are much more complicated than transformation problems, and there has been a rapid development in the area of comparing the peformance of novices and experts on non-trivial problems. All of this research supports the general conclusion that expert problem solving behavior is strongly schema-driven. That is, an expert has a generalized plan or schema for a given problem or class of problems and adapts the schema to solve the current problem. This is a view of problem solving that is very similar to that incorporated into Sussman's (1977) PSBDARB model.

The general paradigm in this research involves comparing the performance of experts and novices on problems that both can solve and then examining the differences in the processes by which they attack the problem. The classical example of this style of research are the studies of Chase and Simon (1973) and de Groot (1966). These studies found that expert chess players differ from good amateur players not in their ability to apply more eficient search and evaluation strategies or to consider a larger number of alternative move sequences, but rather that experts had memorized a much larger number of chess patterns and the "correct", or most favorable, move associated with each pattern. In other words, experts had stored a large number of situation specific schemata.

Another study of expert problem solving behavior is Bhaskar and Simon's (1977) study of an expert problem solver in the domain of engineering thermodynamics. The problems given to the expert subject were various thermodynamics problems from an undergraduate course for which the expert was a teaching assistant. The expert subject's problem solving behavior was striking in its regularity. The subject

began by retrieving one of the few forms of the basic thermodynamics equation and then modifying, or specializing, the equation to fit a particular problem. A means-ends-like process was then used to fill in the variables in the retrieved schema. In fact, the expert used this "thermodynamics schema" even when another approach, with which the subject was presumably familiar, would have made the problem easier to solve. Additional processes used by the subject included the retrieval of additional relevant equations and tables useful for determining the values of specific quantities, the ability to deduce, from key words and phrases, default values (frequently zero) for terms not explicitly mentioned in the problem, and a fairly elaborate procedure to detect errors and mistakes.

Larkin (1977), using a fairly difficult set of problems, has compared the problem solving behavior of novices and experts on various mechanics problems from an undergraduate physics textbook. Larkin examined in some detail the behavior of a single expert subject. She found that this expert constructed hierarchical solution plans, first solving the problem in an abstracted form, then expanding this solution to the level of detail necessary to solve the problem as given. Larkin's main findings are that experts organize their knowledge into "chunks" of related principles and equations, and that they begin problem solving by attempting to find a match between a particular chunk and the problem to be solved. No equations were written until a satisfactory chunk was retrieved. The novices, on the other hand, were more likely to begin writing equations almost immediately. Larkin examined the distribution of times between mentioned equations. She found that for experts, equations were mentioned in bursts, with

same-chunk pairs having very short interresponse times and with longer interresponse times occuring between equations from different chunks. For the novice, the distribution of interresponse times did not differ for equations that were considered to have come from the same or different chunks.

Finally, Hinsley, Hayes, and Simon (1976) have shown that even the solution of elementary high school algebra problems seems to be schema-driven. College subjects were given a collection of problems from a high school algebra textbook. They were asked to classify these problems into a number of subject-determined categories. The classification process appeared to be quite reliable across subjects, and classification seemed to be based on classes of problems as specific as river problems, interest problems, etc. Furthermore, subjects were able to identify the problem class or the solution schema appropriate to a given problem from the first two or three sentences of the problem description.

In summary, current evidence strongly suggests that problem solving in what Bhaskar and Simon (1977) call "semantically rich" domains is strongly schema-driven. Expert subjects use pre-existing plans that are adapted to the solution of the current problem. In all of these studies, however, there is very little evidence or information on the processes by which plans are synthesized.

KNOWLEDGE REPRESENTATION ISSUES

Although plans and planning are terms that are usually associated with research on problem solving behavior, plans have also been discussed in conjunction with comprehension and understanding. This

line of research is particularly relevant since, as we indicated above, problem solving in semantically rich domains may be strongly schema-driven. That is, we feel that the generation of a plan is guided by schemata such as "input-process-output" in a manner very similar to the way understanding of a narrative is governed by macrostructures such as "setting-complication-resolution" (Kintsch and van Dijk, 1975). Although we do not wish to equate planning and comprehension, we feel that the interaction of the comprehension processes with the relevant knowledge structures may be very similar to the interaction of planning processes with task-appropriate knowledge.

Schank and Abelson (1977) have developed a set of concepts that are analogous to many of the concepts developed in this paper. These authors develop three concepts related to the notion of a plan -- scripts, plans, and goals. While these concepts do not map precisely onto the theoretical framework discussed here, primarily because of non-trivial differences between the tasks of planning and understanding, their concepts are closely related to certain structures in our theory.

Schank and Abelson define a script as a "predetermined, stereotyped sequence of actions that defines a well-known situation" (p. 41). We see a fairly direct correspondence between scripts and the detailed plan level of a procedural net. These detailed plans, or scripts, comprise a large part of a skilled software designer's knowledge. These include such topics as sorting, merging, manipulating linked lists, etc. In many cases, solving a software design problem involves a fairly direct mapping of problem elements into a sequence of subgoals at the detailed plan level. Once such a sequence of subgoals

has been correctly defined, the goals can be expanded on the basis of this script-like knowledge.

The concept of a plan as described by Schank and Abelson is somewhat different from the way we have used this term. A plan, to them, is "the repository for general information that will connect events that cannot be connected by the use of a standard script" (p. 70). A plan represents the actions underlying a set of goals; this sequence of actions is more novel and less stereotyped than a script. People do have pre-stored plans, but they are much less detailed than scripts and require that much more information, especially more detail, be filled in. We see a close analogy between our concept of an abstract plan and Schank and Abelson's concept of a plan. In generating an abstract plan, a person will modify some general schema to fit the constraints of the current problem. This schema will be applied in a manner that is somewhat different from the way it has been used to solve previous, similar problems. In refining this plan, the designer may invoke several script-like entities. These schemata will require at most minor changes in order to be used in the current situation.

Schank and Abelson are also concerned with the goals that underlie particular plans. In some sense, the top level node of the procedural net in our theoretical framework represents the goal being used to guide the generation of the plan. Schank and Abelson, however, allow for the existence of several concurrent goals. We admit that it is probably inadequate to assume that we can represent this top level node as simply "solve the problem." For example, as well as having the goal of solving the problem, a designer may also decide to do so in the most

efficient way possible. The existence of multiple, possibly conflicting, goals will be a necessary addition to our theoretical framework and is expected to be a central focus in our future research.

PROBLEM SOLVING AS UNDERSTANDING

Greeno (1977, 1978) has attempted to develop the concept, originally expounded by the Gestalt psychologists, that "problem solving" is what is involved in the process of "understanding." In this brief section, we would like to juxtapose Greeno's ideas with the ideas of Schank and Abelson on understanding and planning in the domain of text understanding. We feel that a reconciliation or unification of these two seemingly different definitions of understanding may well provide some useful insights into the problem solving processes involved in semantically rich domains such as software design.

Greeno (1977) has attempted to develop the Gestalt psychologists' notion of problem solving as a process of understanding in the context of modern theoretical developments in the understanding of natural language (e.g., Schank, 1972; Winograd, 1972). Greeno asserts that "understanding is a constructive process, in which a representation is developed for the object that is understood" (Greeno, 1977, p. 4). Greeno makes it clear that understanding involves the construction of a representation that corresponds to the structure of the actual object being perceived, the sentence being understood, etc. In his discussion, Greeno develops three criteria for good understanding. The first is that good understanding involves achievement of a coherent representation. The second is that the representation generated by an individual should correspond to the actual structure of the object that

is to be understood. The third criterion is that good understanding has occurred to the extent that the to-be-understood object and its components are related to previously existing knowledge (Greeno, 1977, pp. 44-45).

In our introduction, we defined a plan as a hierarchical structure that underlies the sequence of operations necessary to solve a given problem. We argue that Greeno's definition of understanding directs itself to the generation or discovery of the structure underlying the solution of the problem, rather than a simple rote acquisition of the sequence of operations that, in fact, would solve a given exemplar of a class of tasks (cf., Wertheimer's (1945) discussions of "productive thinking" and Duncker's (1945) distinction between "analytic" and "synthetic" problem solving). In his development of a definition of understanding, Greeno includes both the process of understanding a problem as given and the processes involved in the actual construction of a solution.

Schank and Abelson (1977), on the other hand, clearly limited the concept of understanding to the understander's identification of the major subgoals and primary components of the story, the motivation of the actors involved in the story, or the primary subgoals defined by a problem contained in some textual description. They specifically excluded the detailed kinds of problem solving processes that are required to generate a complete plan or solution of a problem. Following Schank and Abelson, we would like to separate the processes of planning and understanding. While we agree that they are closely related and are persuaded in many respects by Greeno's argument that problem solving and understanding are identical processes, we agree

with Schank and Abelson that understanding involves identifying the major subgoals or major elements of the object to be understood.

Thus, in our definition we want to restrict the term "understanding" to those processes that lead a subject to identify the major relevant aspects of a problem description and that enable a subject to bring to bear relevant, general aspects of his or her total knowledge about the problem domain involved. We would like to reserve the terms "planning" and "problem solving" to refer to those processes which, as described by Sussman (1977), focus on issues involving debugging, or modifying, these knowledge structures.

Earlier, we characterized software design as a problem of arrangement. We feel that the process of understanding involves the identification of the major elements of design and computer science knowledge that are relevant to a particular task. In addition, the process of understanding probably leads to an initial specification of how the elements are to be arranged in an ultimate solution to the problem. The remaining planning and problem solving processes dominate a majority of the skilled individual's time; they are the kinds of debugging processes that are involved in fitting together the script-like elements into coherent solutions to a given problem.

In summary, we feel that the process of generating an abstract plan and the process of understanding, as described by Schank and Abelson and others, are in fact very similar. At this level, we have no conflict with Greeno's equating of problem solving and understanding. Clearly, an effective, correct, useful abstract plan must satisfy all three of Greeno's criteria for good understanding. On the other hand, we feel that the process of articulating a detailed

plan that is consistent with the abstract plan is more similar to Sussmann's characteriation of problem solving as a process of debugging, or problem solving as a process of tailoring given elements to fit a specific context.

METHOD OF DATA ANALYSIS

In the sections that follow, we present the results of two experiments. These experiments involved the collection of verbal protocols and written problem solutions. In some cases, we present our representation of a subject's solution. Our representation involves the elements of the solution that were identified by a subject and the relations among these elements.

As we indicated in an earlier section, a software design requires the explicit documentation of subgoals, which are represented as functions, procedures, modules, etc. We identified these subgoals in one of two ways. If a subject used a flowchart representation, we took as subgoals any name or element that was enclosed in a single "box" of the flowchart. If a subject used a "program design language" type of representation, each element (or line) of the design that was later expanded was taken as a subgoal. Our identification of subgoals is consistent with the normal usages of these types of design documentation.

Our identification of the relations among subgoals is also consistent with the normal usages of the types of documentation employed. Hierarchical relations are expressed through indentation, with subordinate subgoals being indented farther to the right than the superordinate subgoals, with explicit numbering of subgoals (e.g., 2.0, 2.1, 2.1.1) and through "calling seguences" and "control structures" (subordinate subgoals are "called" by their superordinates).

The identification of subgoals and their relationships was also aided by some of the subjects' comments. In comparison with the problem behavior graphs presented by Newell and Simon (1972), our form

of representation is somewhat primitive. This representation, however, is intended to be a highly objective summarization of a subject's overt solution which does not include assumptions about the processes involved or more covert elements of the solution. These representations are intended only to be veridical descriptions of subjects' solutions to software design problems.

AN INVESTIGATION OF PLANNING BEHAVIOR

EXPERIMENT 1 -- THE PROBLEM SOLVING BEHAVIOR OF EXPERTS

This experiment involved the collection of long, thinking out loud protocols from three highly experienced computer scientists. The problem given to our experts is shown in Figure 1. The subject is asked to design a page-keyed indexing system. This problem was selected because it is of moderate difficulty, understandable to individuals with a wide range of knowledge of software design, and does not require knowledge of highly specialized techniques that would be outside the competence of our expert subjects. By understandable, we mean that the nature of the task, the purpose of the program to be written, would be clear to even a novice software designer. By specialized techniques, we mean that the design of a useful page-keyed indexing system does not require an expert to have detailed knowledge of exotic techniques that are used in only very specialized areas of computer science.

In our description of the behavior of the three expert subjects, we will roughly classify various segments of the protocol as representing different activities relevant to the construction of a software design. The first part of every protocol contained a discussion of the elements of the problem, pointing out various schemata at different levels that would be relevant to the solution of such a problem. Also, all of our expert subjects discussed design strategies and design techniques. After mention of various alternative design techniques and strategies, the expert would tell us the overall method that he was going to use to solve the problem, and the remainder of the protocol would conform to these stated intentions. In summary,

PAGE-KEYED INDEXING SYSTEM

BACKGROUND.

A BOOK PUBLISHER REQUIRES A SYSTEM TO PRODUCE A PAGE-KEYED INDEX. THIS SYSTEM WILL ACCEPT AS INPUT THE SOURCE TEXT OF A BOOK AND PRODUCE AS OUTPUT A LIST OF SPECIFIED INDEX TERMS AND THE PAGE NUMBERS ON WHICH EACH INDEX TERM APPEARS. THIS SYSTEM IS TO OPERATE IN A BATCH MODE.

DESIGN TASK.

YOU ARE TO DESIGN A SYSTEM TO PRODUCE A PAGE-KEYED INDEX. THE SOURCE FILE FOR EACH BOOK TO BE INDEXED IS AN ASCII FILE RESIDING ON DISK. PAGE NUMBERS WILL BE INDICATED ON A LINE IN THE FORM /*NNNN WHERE /* ARE MARKER CHARACTERS USED TO IDENTIFY THE OCCURRENCE OF PAGE NUMBERS AND NNNN IS THE PAGE NUMBER.

THE PAGE NUMBER WILL APPEAR AFTER A BLOCK OF TEXT THAT COMPRISES THE BODY OF THE PAGE. NORMALLY, A PAGE CONTAINS ENOUGH INFORMATION TO FILL AN 8 1/2 X 11 INCH PAGE. WORDS ARE DELIMITED BY THE FOLLOWING CHARACTERS: SPACE, PERIOD, COMMA, SEMI-COLON, COLON, CARRIAGE-RETURN, QUESTION MARK, QUOTE, DOUBLE QUOTE, EXCLAMATION POINT, AND LINE-FEED. WORDS AT THE END OF A LINE MAY BE HYPHENATED AND CONTINUED ON THE FOLLOWING LINE BUT WORDS WILL NOT BE CONTINUED ACROSS PAGE BOUNDARIES.

A TERM FILE, CONTAINING A LIST OF TERMS TO BE INDEXED, WILL BE READ FROM A CARD READER. THE TERM FILE CONTAINS ONE TERM PER LINE, WHERE A TERM IS 1 TO 5 WORDS LONG.

THE SYSTEM SHOULD READ THE SOURCE FILES AND TERM FILES AND FIND ALL OCCURRENCES OF EACH TERM TO BE INDEXED. THE OUTPUT SHOULD CONTAIN THE INDEX TERMS LISTED ALPHABETICALLY WITH THE PAGE NUMBERS FOLLOWING EACH TERM IN NUMERICAL ORDER.

A NULL SOURCE FILE INDICATES THAT PROCESSING IS COMPLETED. ERROR MESSAGES AND A TERMINATION MESSAGE SHOULD BE WRITTEN TO THE OPERATOR'S CONSOLE. EACH COMPLETED INDEX IS TO BE STORED ON DISK FOR LATER LISTING.

Figure 1. The Page-Keyed Indexing Problem

these initial segments of the protocol involved the subjects retrieving and instantiating a variety of knowledge structures, ranging from information about design strategies and resource allocation policies, to schemata that described particular subdesigns for aspects of the potential solution. In terms of classical work on problem solving, the behavior in these initial segments of the protocols could be described as preparation.

The remainder of each of the protocols concerned the construction of the actual software design. Various elements of our hypothesized plan structure appeared in the protocols of all subjects. The subjects differed in the order in which they constructed this hypothesized structure. All subjects seemed to understand that the completed design would be a structure like a procedural net, varying in levels of abstraction, with the transition from one level to the next characterized by terms like 'step-wise refinement' or 'increasing levels of detail'. Subjects differed widely in the processes they used to construct this structure. Furthermore, they were quite explicit about the processes they would use to synthesize the underlying structure or procedural net.

One final aspect of subjects' planning behavior should be mentioned. All of our experts rapidly recognized that various elements of the solution to the page-keyed index problem involved algorithms that were well understood and that in many cases optimal algorithms were known in the literature. Our subjects found the retrieval of such algorithms difficult, because they had not bothered to commit all of the details of the algorithm to memory. In normal circumstances they would look up the details in an appropriate reference. This is one

artificial aspect of the design task as given to the subject, and attempts to retrieve relevant details consumed an inordinately large part of the protocols.

In summary, the protocols could be partitioned into two phases. The first phase we will characterize as preparation, and the second phase we will call planning. It is in the planning phase that the subjects developed their actual design. The preparation phase involves a careful reading and summarization of the problem description, discussion of relevant techniques, and some discussion of the design techniques to be used in solving the problem given in this particular task.

The Behavior of S2. S2 is a Master's degree candidate in computer science who is employed as a systems programmer. S2 has a great deal of experience with various text processing applications, although S2 had not designed or written any type of indexing program. In addition, S2 is interested in software design methods and techniques and regularly reads the technical literature in this area.

The preparation phase of S2's protocol was relatively brief and contained three principal elements: a brief statement of design techniques, a rather clear statement of a schema within which he could analyze the given problem into the elements necessary to derive a software design, and a summarization of the problem as originally given. S2 states that he intends to use a program design language of his own that incorporates a large number of structured programming techniques and that shows a lot of influence from the programming language PASCAL. He also makes it clear that his design strategy is to proceed in a top-down breadth-first fashion. S2 then given a very

clear statement of his most abstract design schema:

> "One thing that I believe absolutely has to be done is to write a complete, detailed specification of what I consider to be the user interface what the input is going to be and what the output is going to be. Once that's done, I then attempt to analyze the problem of taking the input data and transforming it into the output results in terms of what are the primitive objects that I need to have."

We interpret this quote to instantiate a very general schema of the form 'initialize-input-process-output'. We have given a graphical representation of this schema in Figure 2. We feel that S2 was instantiating his most abstract or high level design schema, and that it would serve as a template for construction of the abstract plan as well as more detailed elements of the final design.

The final element of S2's preparation phase was a careful analysis of the problem as given and a summary of his understanding of the problem. We show S2's analysis of the problem in Figure 3. Note that it is simply a summarization of the major elements of the problem statement and could be derived by an individual who was planning to construct a page-keyed index by hand as well as someone who was planning to write a computer program to accomplish the same task.

S2 constructed the design to carry out the page keyed indexing task in a strictly top-down breadth-first fashion. He used a program design language to state the major elements of the design. S2 defines his design language in the following fashion:

> "The items that I have on each line correspond to the functions to be performed...and can be thought of as procedure calls, so that at this point, my high-level design is complete, and I now proceed to expand and elaborate the procedures and modules which I have identified at this level."

The statement above characterizes the nature of the software design language; it was made after the subject has articulated the abstract

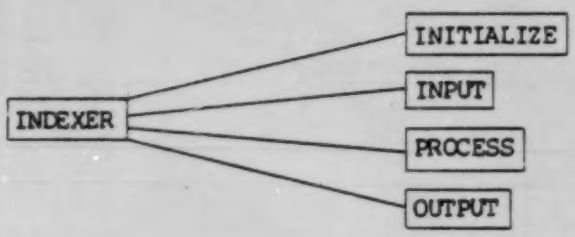


Figure 2. Abstract Design Schema Used by S2.

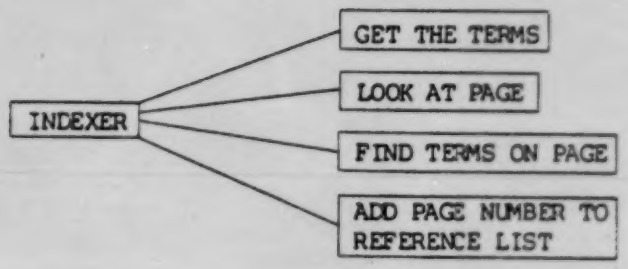


Figure 3. Abstract Problem Schema Used by S2.

2 though 6 in Figure 4, where we present S2's final design. Note that S2 considers the high-level, or abstract, design to be complete at this stage. He then proceeds to articulate the details of the major elements of the design that he has outlined. S2's summary is also consistent with our assumption that each level of detail is a complete solution to the problem. At this level, of course, the plan identified consists of the major subgoals that are to be accomplished, but the lack of detail prevents this level from being considered an implementable solution. The remainder of S2's protocol is concerned with expanding these goals into a detailed plan (which is the last level shown in Figure 4), and then finally to the lowest level, the operation sequence level.

We will only consider one part of the expansion into the lower levels of the detailed plan. The case we will consider is the expansion of the detailed plan 'INSERT TERM IN ORDERED TABLE' which is the successor of the abstract plan 'READ TERMS'. (See Figure 4.)

The expansion of this detailed plan took a great deal of time, about 30-40 minutes in a protocol which took approximately 3 hours, and the subject was becoming very frustrated at this point. He commented that:

> "I feel like I'm getting slightly bogged down in the Insert procedure. It is one that I've written probably 5, 6, 7 times before...Since I, as I am reproducing it exactly, correctly from memory and it takes a great deal of effort, so what I will do is to put down that the node is to be inserted by a classical scheme and simply assume, at this point, that this will be a reference to an existing algorithm."

The subject was attempting to retrieve and reproduce a previously written routine for inserting nodes in an ordered table. At this point, planning effectively stopped and the subject's efforts were

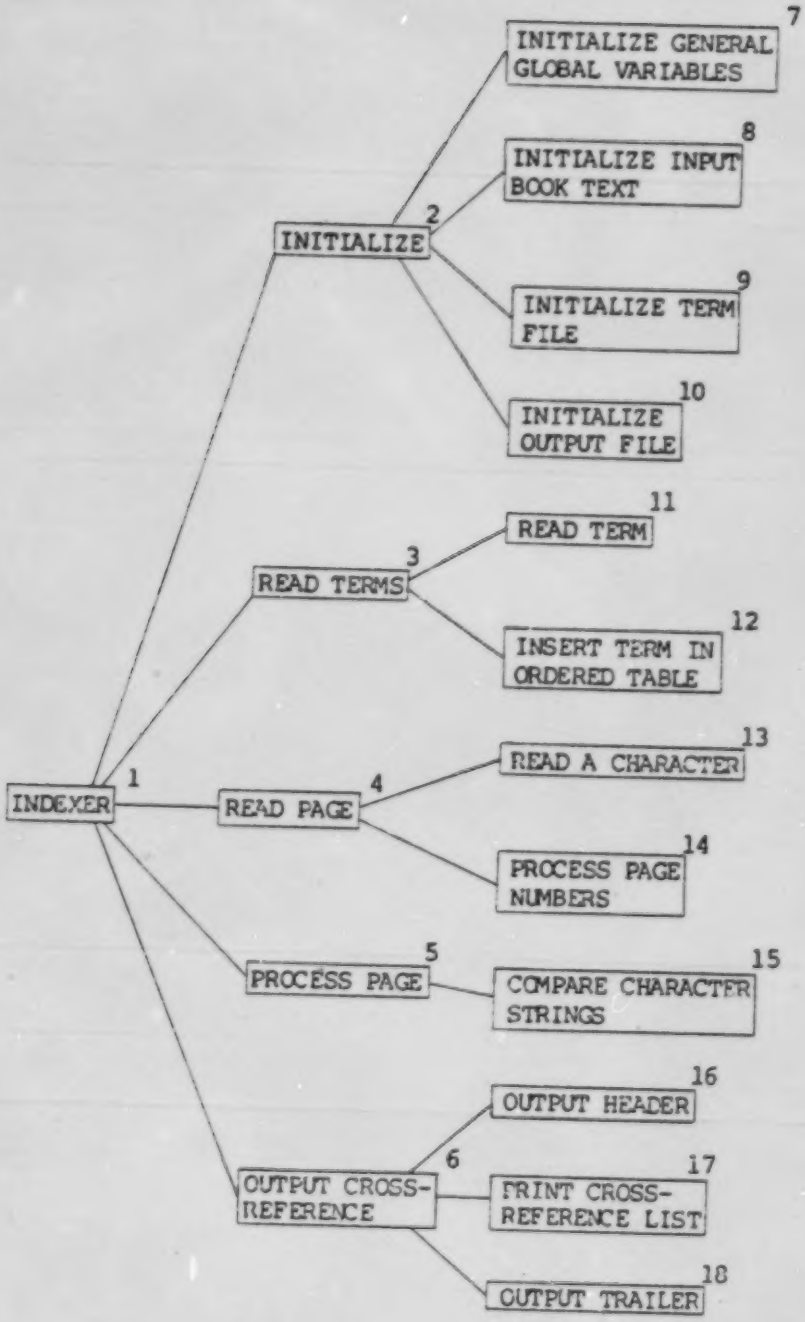


Figure 4. S2's Final Design

directed to the memory retrieval task. In fact, note that this process apparently required so much effort that the subject eventually determined that this exercise was unprofitable and decided to use a "classical scheme", which the subject had developed and used previously, and returned to planning in other areas of the detailed plan level. Notice that the existence of this routine was part of the subject's software design knowledge, but the actual details of its operation were not sufficiently well integrated that they could be easily retrieved.

We argue that this segment of S2's protocol provides strong support for our characterization of the planning process at the detailed plan level of the procedural net to be very closely analogous to Sussman's (1977) characterization of problem solving as the debugging of almost right plans. Planning at this level involves the novel combination of well known elements, and these elements, which a subject must retrieve from memory or from other sources, form the major part, if not all, of the detailed plan and operation sequence levels of the network. The top levels of the abstract plan involve the decomposition of the problem into its major elements. The top levels of the detailed plan are the decomposition of the elements of the abstract plan into known algorithms or schemata for implementing various subparts of the task ultimately to be accomplished. These schemata are then retrieved and articulated in the context of the problem to generate the remainder of the detailed plan level and the operation sequence level of the network.

This protocol provides clear support for the theoretical framework outlined in the previous section. The design was constructed with

clearly defined levels and was expanded in a top-down, breadth-first manner. This form of expansion was readily apparent both in the subject's comments and in the written solution. S2 numbered, on separate pages, the modules or subroutines that were developed and it was from this numbering that we derived the hierarchical structure of our representation of this design, as shown in Figure 4. The results obtained from the other two experts, however, differed from this protocol in several aspects.

The Behavior of S3. S3 has degrees in physics and electrical engineering and about 15 years of programming and design experience. Although S3 is aware of current developments in design technqiues, this subject has no formal training in this area.

Again we can partition S3's protocol into two major segments: the preparation phase followed by the construction of the actual plan. The preparation phase of S3's protocol is quite long and discursive. It includes an analysis of the problem as given, discussion of various relevant pieces of software design information, and discussion of particular algorithms or techniques that might be useful later in solving the problem. We characterize the preparation phase of S3's protocol as being 'opportunistic' in the sense used by Hayes-Roth and Hayes-Roth (1978).

In contrast to the protocol of S2, the protocol of S3 is much more difficult to interpret. S3 used a flowchart as the primary form of representation and the expansion of the design was frequently "interrupt driven." As we will show below, the overall design does show a top-down expansion, first to an abstract plan, then to a much more detailed plan, and finally to the level of individual actions.

The details of this expansion, however, differ from S2's in several important aspects.

First, when elaborating the design at the abstract level, S3 followed the description of each abstract node with a brief sketch of the potential descendants of this node. This enumeration seemed to be driven by a schema for generic nodes of this type. For example, when considering the "OUTPUT INDEX" node (see Figure 5) S3 mentioned headings, terms being of variable lengths, different numbers of references for each term, and so on. The items mentioned seemed to be guided by this subject's experience with writing output modules for other programs. Each item was resolved in one of three ways; 1) the item was seen as presenting no problem (an existing schema was adequate); 2) the item was seen as presenting a problem and one or more potential solutions were discussed (the schema needed a correction in order to be applicable); and 3) the item was seen as presenting a problem, but no solution was forthcoming (no appropriate schema was found). In this latter case, the item was flagged as a potential problem and further consideration was postponed until the next level was expanded.

The second strategy that seemed to drive S3's behavior, both in this "pre-expansion" mode, and later, when actually elaborating the detailed plan, was a "check for errors" rule. Whenever S3 generated a data structure, read data, or instantiated a fairly standard procedures (e.g., "compare words"), S3 immediately considered possible errors or anomalous cases; for example, end of file, [illegible] numbers, or extra blanks in terms. Corrections were dealt with in essentially the same manner as described above. Several times, the solution to a

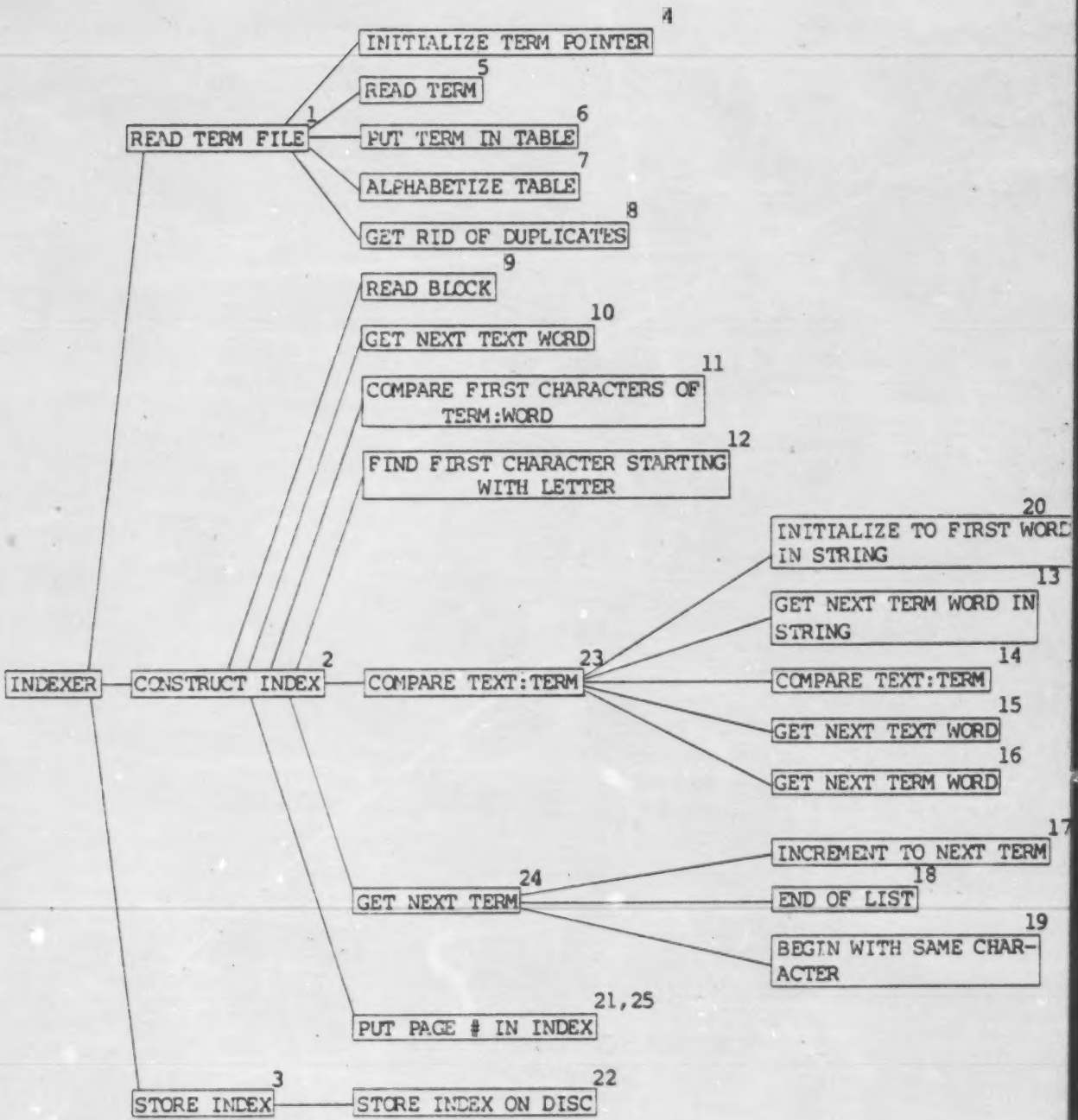


Figure 5. S3's Final Design

potential error required modification of a node that was the descendant of a node other than the parent of the current node. Although S2 exhibited some of this "check for errors" type of behavior, this strategy is much more pronounced in the protocol of S3.

S3 also shows evidence of two aspects of expanding a procedural net, or design, that are not found in S2's protocol. First, in several cases, S3 considers two alternative ways to produce a given result and delays choosing between them until some later node produces a preference for one of the two methods. This is the software design analog of Sacerdoti's (1975) "use imaginary objects" method to delay binding of variables to values until a clear preference for one of the candidates is established. Second, in S3's protocol, there are several instances of backtracking. They are resolved either by revising or augmenting an earlier node at the same level (e.g., adding an earlier node to satisfy a precondition), or one case, by deciding that the expansion was not working and scrapping all of the successors of a given node and attempting a different decomposition.

Our representation of S3's design is presented in Figure 5. Recall that S3 used a flowchart as the means of representing the design. In this type of representation,"flow of control" and "control structure" are emphasized more than they are in other forms of representation, such as that used by S2. Since "decision points" were presented as separate "boxes" in S3's design, we have included them in Figure 5. It should be noted, however, that S3 explicitly stated that these elements of the design (nodes 18 and 19 in Figure 5) were not to be expanded further and that no explicit instructions need be given about these elements to those who would implement this design. Also

excluded from further expansion were two "initialization" routines (nodes 4 and 20).

At first glance, S3's design does not appear to have been expanded in a top-down breadth-first manner. Notice that nodes 13 through 16 and node 20 were expanded before the predecessor (node 23) and that nodes 17 through 19 were generated before node 24. Although this appears to be an example of bottom up expansion, S3's comments indicate that this was an oversight rather than a deliberate form of expansion.

Throughout the experimental session, S3 expressed the philosophy that design decisions should be deferred to the lowest possible level, and referred to this strategy as "let George do it." After S3 had generated nodes 13 through 20, this oversight was noted and the subject commented:

> "And here I find that I do this, probably because of the way that I approach this, eventually this 'let George do it' rather collapses, and as I look back on what I have tentatively written down, I find that I put things, indicated the thing should be on one level, where conceptually they do not belong, because of the detail involved with the type of stuff that it's done. It just doesn't belong there. It should be more by itself, that is to say, further down the pike."

At this point, S3 introduced nodes 23 and 24 and indicated that they were the successors of nodes 13 through 20.

The generation of node 20 is also contrary to a strict top-down breadth-first expansion. This is a clear example of backtracking, as was mentioned earlier. While expanding the node to "GET NEXT TERM", S3 noticed that some initialization was necessary for the "COMPARE TEXT AND TERM" node and inserted an initialization routine. Notice, however, that this addition was at the same level of the design as the expansion that caused this backtracking behavior. S3's concern with

designing in definite levels of detail is apparent in the comment:

> "Now we have several places that we could go to elaborate further what's in these boxes, but I find it easier, until I get lost in it, well not really lost, so much as really embroiled in a particular problem, to stay at the same level which would mean, at this point, rather than elaborating on the boxes on the page, here, READ TERMFILE, to go back to the higher level, move on to the next box, which is constructing the index, which is really the implementation of that box at the same level."

Clearly S3's expansion of the design is by levels of detail. The only major aspect of this form of expansion that appears to be contradictory to the theoretical framework outlined earlier is that this expansion does not appear to be strictly top-down and breadth-first. As we indicated above, however, this is primarily due to an oversight on the subject's part, and the subject corrected this oversight as soon as it was detected. Although the forms of expansion observed in S2 and S3 appear, on the surface, to be different, this difference appears to be due more to the ability to successfully execute a common strategy for expansion rather than to fundamentally different strategies. The behavior of S5, however, may indicate a different strategy.

Behavior of S5. S5 is a doctoral student in computer science who returned to graduate school after several years of experience as a professional programmer and designer. S5 is extremely knowledgeable of the literature of both the applied and theoretical aspects of computer science, in particular, the area of software design. Like the other two protocols, S5's can be partitioned into preparation and planning phases. S5 gave us one of the longest preparation phases. In this phase, he articulated a particular theory of design. He then proceeded to construct his design in the manner that he had described during the

preparation phase. As will be seen in the remarks below, and in the description of S5's plan, the principle component of S5's approach was an emphasis on efficiency of the program to be ultimately derived from the design. That is, S5 evaluated the designs primarily on the basis of storage requirements and execution speed.

S5 expressed the general philosophy that software designs should not be done "from scratch." Since optimal designs for a large variety of functions (e.g., sorts, merges, etc.) have been developed, S5 expressed the belief that the designer should find such designs and incorporate them into the overall design, rather than "reinvent the wheel" and, thus, incorporate sub-optimal designs for the necessary functions.

When presented with the design task, S5 commented that optimal designs for all of the functions that would probably be needed could probably be found in reference books, especially in the series of books by Knuth (e.g., 1968) on the "art of programming." Further, S5 noted that a recent journal contained a proof, related to searching tree structures, that led to the development of an optimal search strategy, and that he would prefer to review this article before beginning the design, since its incorporation would greatly aid overall efficiency. For the purpose of this experiment, however, S5 agreed to perform the design task "from scratch." Since our emphasis is on planning behavior, rather than on optimal software design, we do not feel that this shift in preferred strategies on the part of the subject adversely affects our results. In summary, although S5 was not able to incorporate previously developed optimal designs into the overall design, there was still an extreme emphasis on efficiency that was not

observed in the previous two subjects, This emphasis produced a type of behavior that was different from the other expert subjects.

Two procedural changes were adopted prior to the collection of this protocol. First, the problem was simplified somewhat. Our first two expert subjects spent a great deal of time considering design alternatives and questions that were closer to the actual implementation than to the plan for solving the problem. This included such considerations as how to handle multiple instead of single blanks between words, how to distinguish between a word hyphenated at the end of a line and a word that contains a literal hyphen (e.g., line-printer), etc. Since such considerations tended to cause the subjects to spend a great deal of time on small, isolated aspects of the design rather than on the planning aspects of design, we rewrote the problem specification to eliminate those elements that were encountered by S2 and S3. The simplified version of this problem is shown in Figure 6. Except for details, this problem statement corresponds exactly to that used earlier.

Second, we presented S5 with a list of "primitives" and requested that the design be expressed in terms of these primitives. The subject was told to assume that these primitives, which consisted of various functions and routines, could be used in the design without being described in detail. This was intended to prevent subjects from attempting to retrieve the solutions to well known functions (see the discussion accompanying Figure 4 in S2's protocol) and also to provide some commonality in the language used by subjects to describe their designs. These primitives, which are shown in Figure 7, were obtained, in part, from the protocols of the other expert subjects and also from

PAGE-KEYED INDEXING SYSTEM

BACKGROUND.

A BOOK PUBLISHER REQUIRES A SYSTEM TO PRODUCE A PAGE-KEYED INDEX. THIS SYSTEM WILL ACCEPT AS INPUT THE SOURCE TEXT OF A BOOK AND PRODUCE AS OUTPUT A LIST OF SPECIFIED INDEX TERMS AND THE PAGE NUMBERS ON WHICH EACH INDEX TERM APPEARS. THIS SYSTEM IS TO OPERATE IN A BATCH MODE.

DESIGN TASK.

YOU ARE TO DESIGN A SYSTEM TO PRODUCE A PAGE-KEYED INDEX. THE SOURCE FILE FOR EACH BOOK TO BE INDEXED IS AN ASCII FILE RESIDING ON DISK. PAGE NUMBERS WILL BE INDICATED IN THE FORM *NNNN WHERE "*" IS A MARKER CHARACTER USED TO IDENTIFY THE OCCURRENCE OF PAGE NUMBERS AND NNNN IS THE PAGE NUMBER. "*" IS A "RESERVED" CHARACTER AND IT WILL NOT APPEAR ANYWHERE ELSE IN THE TEXT.

THE PAGE NUMBER WILL APPEAR AFTER A BLOCK OF TEXT THAT COMPRISES THE BODY OF THE PAGE. THE PAGE NUMBERS WILL BE IN ASCENDING ORDER, BUT NOT NECESSARILY IN SEQUENTIAL ORDER. A BOOK MAY CONTAIN PAGES THAT CONSIST OF ILLUSTRATIONS OR FIGURES. SINCE SUCH PAGES ARE NOT TO BE INDEXED, THEY ARE NOT INCLUDED IN THE SOURCE FILE. NORMALLY, A PAGE CONTAINS ENOUGH INFORMATION TO FILL AN 8 1/2 X 11 INCH PAGE. EACH PAGE OF TEXT IS STORED AS A SINGLE RECORD. EACH WORD IS PRECEDED BY A SINGLE BLANK AND MAY BE FOLLOWED BY A SINGLE PUNCTUATION MARK. IN ADDITION, SINGLE WORDS DO NOT CROSS PAGE BOUNDARIES AND THERE ARE NO HYPHENATED WORDS.

A TERM FILE, CONTAINING A LIST OF TERMS TO BE INDEXED, WILL BE READ FROM A CARD READER. THE TERM FILE CONTAINS ONE TERM PER LINE, WHERE A TERM IS 1 TO 5 WORDS LONG. THE TERM FILE WILL BE INPUT IN ALPHABETICAL ORDER. ALL TERMS START IN COLUMN 1 OF THE CARD AND WORDS ARE SEPARATED BY SINGLE BLANKS.

THE SYSTEM SHOULD READ THE SOURCE FILE AND TERM FILE AND FIND ALL OCCURRENCES OF EACH TERM TO BE INDEXED. THE OUTPUT SHOULD CONTAIN THE INDEX TERMS LISTED ALPHABETICALLY WITH THE PAGE NUMBERS FOLLOWING EACH TERM IN NUMERICAL ORDER.

Figure 6. Simplified Version of the Page-Keyed Indexing Problem

a second experiment to be described later. These differences, however, cannot explain the large differences between the behavior of S5 and that of the other expert subjects.

In the majority of the preparation phase, S5 considered the design problem as a whole, identified the constraints that would have to be observed while doing the design (file sizes, record lengths, etc), identified various parts of the design, and produced algorithms for, or other descriptions of, these parts. The product at the end of this phase was several pages of notes, rather than a completed design. The documentation of the complete design was the second segment of S5's protocol.

The final design produced by this subject is shown in Figure 8. This representation is taken from the second segment of S5's protocol, in which the design was actually documented. This documentation presents the design in a strictly top-down, breadth-first manner, with few exceptions, First, S5, like S3, did not explicitly describe a single top level node, such as the "INDEXER" node incorporated by S2. Since, in the design as presented, the top level node, or "main routine" would contain only the control, or calling sequence, for the routines at the next lower level, S5 apparently assumed that this information did not need to be explicitly documented. This is a reasonable assumption since subjects were told that the designs would be implemented by "competent" programmers and the specification of the calling sequence, which was sequential in this design, should be within the programmers' level of ability. Similarly, S5 did not completely expand all of the routines, or nodes, that were listed. Rather, it was assumed that the programmers could perform certain functions, such as

ACTIONS	OBJECTS	QUALIFIERS
INITIALIZE	VARIABLE	NEXT
INPUT	FILE	BEFORE
GET	RECORD	AFTER
READ	WORD	FIRST
WRITE	TABLE	LAST
OUTPUT	LIST	PREVIOUS
COMPARE	PAGE	CURRENT
INSERT	CHARACTER	
DELETE		
EXCHANGE		
COPY		
SORT		
MERGE		
CREATE		
CONCATENATE		
MATCH		
PROCESS		
CALL		

Figure 7. List of Design Primitives Used by S5.

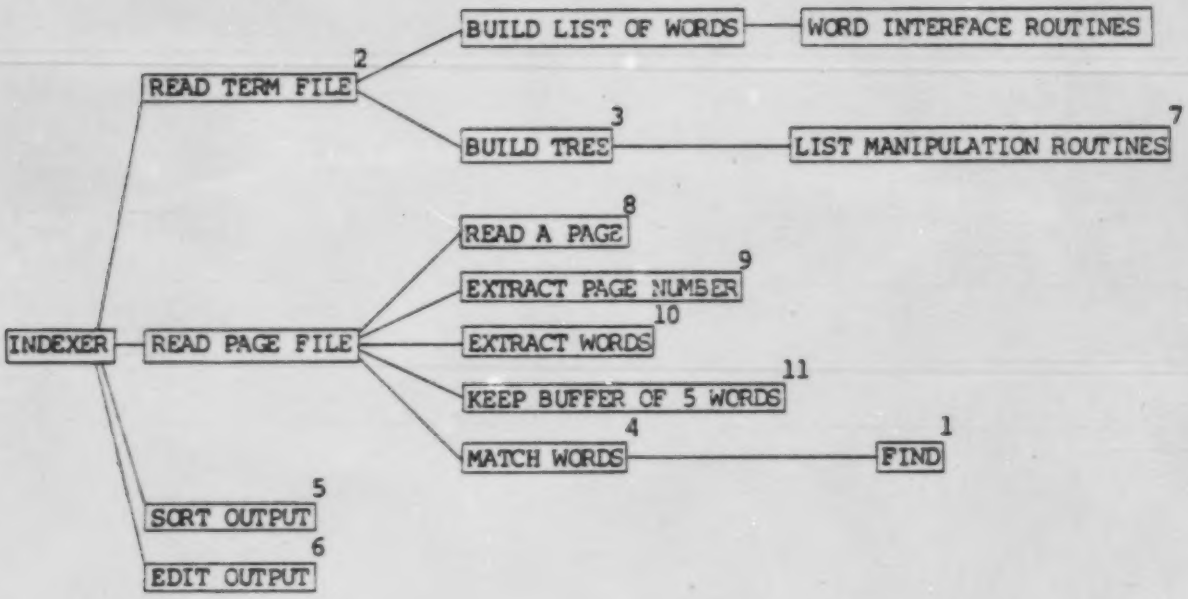


Figure 8. S5's Final Design

detailed instructions. Third, S5 presented the programmers with a far more detailed description of the data structures involved than did the other two expert subjects. This is most probably due to the fact that he had designed a rather complex data structure, and the construction of this structure was not assumed to be within the competence level of the eventual programmers.

When producing the design documentation, S5 expressed the philosophy that "designs should always be presented to programmers in a top-down manner so that they can understand it." This philosophy is clearly evident in the structure of the design presented in Figure 8. S5, however, does not apply this top-down philosophy to the process of constructing the design. This subject contended that designs should "never be done top-down."

The rationale behind this general approach to doing design tasks is a strong emphasis on efficiency and optimality. S5 equated top-down design with doing the design "from scratch." As was indicated above, this subject was aware that optimal designs for a variety of functions frequently encountered in software design tasks were available in reference books. S5 further explained that a design developed in a top-down manner would probably result in modules at the lower levels that could not be efficiently implemented in assembly language. This emphasis on efficiency caused S5 to only briefly consider a high level solution to this problem and then isolate the single component of the completed design that was judged to most affect overall efficiency.

Intially, S5 considered the structure of the term file and page file (the inputs to the problem) and the structure of the output file (see Figure 6 for the description of the design problem). This subject

rapidly determined that the most important aspect of this problem, in terms of efficiency, is the comparison of words in the term file with words in the text file. Next, S5 designed a routine to perform this process. This routine suggested that efficiency could be increased if the term file were re-organized. The resulting file structure, which involved a tree structure in which the top level was all words that appeared first in terms, the next level was all words appearing second, etc. (a doubly linked list) was developed next. Considering this data structure suggested, to S5, that the routine to compare words could be made more efficient if it were recursive.

The resulting routine, called "FIND", is shown in Figure 8. In Figure 8, we also indicate the order in which each routine was developed during the course of doing the design. Routines that are not associated with a number were included in the documented version of the design, but not mentioned during the initial development of the design. Notice that the FIND routine is at the lowest level developed, at what we would characterize as the "detailed" plan level. The next several nodes considered seemed to be chosen because of their close relationship to the FIND routine (i.e., BUILD TREE of terms and MATCH WORDS). Thereafter, the design was expanded in a basically top-down manner.

Clearly, the mode of expansion observed in this protocol differs from that observed in the protocols of S2 and S3, although there are many similarities in the final designs. The difficulty of interpreting S5's protocol in terms of the theoretical framework presented earlier led to a consideration of why such a mode of expansion would be used. For example, is it possible to articulate the lower levels of a plan

before considering the higher levels or is it only the case that this particular subject omitted explicit discussion of the higher levels of the plan before considering the lower levels?

Although we feel that plans are, in general, expanded in a top-down fashion, we hypothesize two reasons why some other form of expansion may be adopted. Both reasons are concerned with a generic schema for the type of problem presented. In the first case, a well developed schema is present, and in the second, no schema exists. This schema represents a previous integration of the relevant software design and problem specific knowledge. Since the problems we are dealing with are essentially "common sense" problems, we do not expect large differences in subjects' problem specific knowledge. Similarly, little difference should be observed, in subjects of comparable experience levels, in general knowledge of computer science and software design.

If a subject, through previous experiences, had developed a general schema for a given class of design problem it is unlikely that overt planning behavior would be observed. Rather, a more general plan could be retrieved from memory and the necessary corrections made to this schema to solve the current problem. Consider, for example, the task of constructing a page-keyed index with pencil and paper. In this case, most subjects would likely turn to the first page of the text and list the terms found there. This behavior would be immediate and little, if any, explicit consideration of plans for this behavior would be apparent. As we have seen above, however, and as some reflection makes obvious, a great deal of planning behavior is necessary. In a "pencil and paper" environment, however, a well learned schema exists,

and this schema is quickly retrieved and applied. It seems reasonable to expect that a similar phenomenon would occur whether the solution of the task were to be expressed as a software design or implemented in some other domain, if the corresponding schema existed. If this schema indicates, as it apparently did for S5, that some element of the plan was more essential to overall success than other elements, then we would expect this element to be considered first.

In the second case, a subject may concentrate on the lower levels of a plan because no schema exists and, in addition, the subject either lacks the appropriate problem related knowledge or is unable to integrate, even at a very high level, his software design and his problem related knowledge. In this case, the subject does not actually understand the problem to be solved, in the sense that well structured definitions are formed for the elements of the initial and goal states of the problem. Considering how some element, or function, could be implemented, in this case, represents a subject's attempt to form a better structured definition of the given problem. It is only after this definition is satisfied that actual planning of a solution can commence.

In the protocol just presented, we feel that it is the first explanation that explains the initial expansion of a lower element of the completed plan. S5's emphasis on efficiency and the existence of a well-developed schema prompted the identification and expansion of the subproblem that, in terms of the final design, most determined overall efficiency.

The Behavior of Expert Subjects: Summary. In the sections above, we have considered, in some detail, the individual behavior of three

expert software designers on a single problem. In this discussion, we have emphasized identifiable differences in behavior. In this section, we will briefly consider the commonalities.

First, we assume that all three subjects share common background knowledge about the problem, and that this knowledge is organized as we have considered it in conjunction with discussing S2's protocol (see Figure 2). In addition, we assume that there are few differences in the form of the initial integration of relevant knowledge structures. Notice that the final designs produced by all three subjects can be described, at the top levels, as 'read terms, read page, find terms, build index'. Nor do we assume that these subjects differed substantially with respect to their software design knowledge. Although S5 was the only subject to incorporate "sophisticated" data structures and recursive procedures, the other subjects are aware of these techniques, but did not consider them appropriate, or necessary, for this problem.

We argue that the only substantive differences across these subjects' is due to the existence of previously developed, generic problem schemata. This is most pronounced in the protocol of S5. Recall, however, that S3 engaged in a type of "interrupt driven" behavior, while S2 did not, suggesting that the generic schema possessed by S3 was not quite as well developed as that of S2, whose expansion was more systematic.

Clearly, there is much more information in these extensive protocols than we have presented in this brief discussion. At this point, the most significant conclusion that can be drawn from the protocols is that the ability to construct plans, and the method by

which plans are expanded, is a function of previous experience. In itself, this conclusion is not surprising. Obviously, previous experiences affect the ability to construct plans for a given class of problems. Less obvious, however, is the question of how experience affects the construction of plans. We will consider this question in the following section, where we discuss the behavior of less experienced subjects.

EXPERIMENT 2 -- ALTERNATIVE METHODS OF PROTOCOL COLLECTION

One purpose of this experiment was to examine alternative methods for the collection of experimental protocols. With one of our experienced subjects (S5), we introduced a list of primitives and required that the lowest level of the design be expressed in terms of these primitives. This manipulation, which did not appear to significantly alter the behavior of this subject, was an initial exploration of the use of "constrained" protocols. In the present experiment, we compare written and verbal (or oral) protocols.

Although many investigators have advocated that thinking aloud protocols are the preferred method for the study of problem solving behavior (e.g., Newell and Simon, 1972), such protocols are difficult to analyze. They first have to be transcribed, which can be very time consuming. Next, they must be transformed into some form of problem-behavior graph (viz,. Newell and Simon, 1972). A problem-behavior graph is a theoretical interpretation of a subject's protocol into a set of knowledge states and the sequence of transitions between knowledge states. This is a subjective process, and the most valid criticism of thinking aloud protocols is that they are very difficult to reliably and objectively transform into some form of problem-behavior graph.

There are numerous reasons for using thinking aloud protocols. First, they provide a very rich source of information about a subject's step-by-step processes in the course of solving a problem. Second, the recording of protocols does not bias a subject towards any particular method of solution and does not interfere with the solution process. In our current research, however, we are attempting to develop

alternative procedures for the analysis of problem solving behavior. Ideally, these procedures would provide us with much of the same information that could be obtained from a verbal protocol, but would be much simpler to analyze.

Subjects. Subjects were undergraduates enrolled in an "assembly language" course at the University of Colorado - Boulder. Most subjects were third year students and all had completed a mimimum of two computer science courses. The mean number of courses completed by this sample of 10 subjects was 2.8. Subjects were paid $5.00 for their participation in this experiment. They were randomly assigned to one of two experimental conditions, with five subjects in each condition.

Procedure. Subjects were given instructions appropriate to the assigned experimental condition. Those in the "verbal protocol" condition were instructed to solve the design problem and to "think aloud" while doing so. These subjects also produced written versions of the solution. Subjects in the "written protocol" condition were instructed only to provide written solutions. All subjects were given the simplified version of the page-keyed indexing problem shown in Figure 6.

After completing their designs, subjects were instructed to write a summary of their design and also a summary of the techniques and procedures used to produce the design. Finally, they all completed a questionnaire describing their background and experiences in computer science.

Analyses. Our initial analysis considered only the design summaries provided by subjects. The questions of primary interest were to determine how much useable information was contained in such

summaries and to determine if there were any obvious differences as a function of experimental condition.

Six of these summaries (three from each experimental condition) emphasized the higher level components of the design and the interactions or relations among these components. In general, the content of these summaries were very similar to the initial decompositions, which we characterize as the abstract plan level, of the expert subjects. Four subjects either reiterated the entire design or concentrated on the functions and procedures involved in the design without specifying how these elements were related.

We initially expected that design summaries would focus primarily on the abstract plan level developed by these subjects. This was true, however, in only 6 of the 10 cases. This expectation was based, in part, on the finding that subjects in text comprehension studies produce summaries that are consistent with the macrostructure underlying the text (cf., Kintsch and van Dijk, 1978). In text comprehension studies, however, subjects generally have well-developed schemata for the types of text that are presented. Subjects in this experiment, however, may not have well-developed schemata for software design problems. We will return to this issue later when we consider the designs produced by these subjects. In general, therefore, design summaries did not produce consistently useful information, although some of the summaries did reflect subjects' abstract plans.

Examples of design summaries that were judged as presenting reasonably well-defined abstract plans are presented in Figures 9 and 10. Notice that these summaries correctly identify the major functions that must be to be accomplished, but do not provide any detail about

```
     Housekeeping and Storage Initialization
     Get terms to index
WHILE NOT END OF SOURCE FILE
Get & Parse Source Line
     Do Until Page Delimiter Found
          If Source Word in Index
               Then Increment Count of Terms
          END-UNTIL
          Get Next Page First Line
          Count Possible Stray Index Terms
          If terms on this page
               Store Page #
               Increment Page #
        END-IF
END-WHILE
```

Figure 9. A Design Summary

```
This is a top down design that deals
only with the information that is known at
one time.
     First a table of terms is formed
     Then one page is accessed and
          one word at a time is broken off
          and looked for in the table
          If it is found the rest of that
          term is looked for
               If a whole term is found then
               a flag is inserted in the table
               for that term
     When the page number is found
          it replaces all current special
          symbols with the page number just found
          When the last page has been scanned
          then table is output.
```

Figure 10. A Design Summary

Two files are created initally (TERM) and (STORE)

a term is read from a card and placed into the file (TERM). This is repeated until all terms have been read.
Now text is read from file (SOURCE) looking for matches of terms (any occurrences of any term). If a term is encountered that term is stored onto file (STORE). This continues until an * is encountered. When the * is found, the next four characters are read into a variable (PGNUM) We rewind (STORE) and proceed thru it one term at a time. As each term is read, its corresponding entry in (TERM) is located, and PGNUM is inserted in appropriate text format immediately following it. This process continues until (STORE) is exhausted, at which time we clear (STORE), PGNUM and begin reading from (SOURCE) again. After (SOURCE) is exhausted, we simply print out the file (TERM).

Figure 11. A Design Summary

are to be accomplished. In Figure 11, we present a design summary that was judged not to represent an abstract plan. In this case, some procedural detail is involved, and the major functions that the design is to perform are not as readily identifiable as they are in Figures 9 and 10.

We were also interested in determining whether the content of the design sumamries was affected by the experimental conditions. Two of us, who were unaware of the assignment to experimental conditions and who had not yet seen the actual designs, attempted to sort the summaries. The content of these summaries, although far from uniform, provided no information that would result in an accurate classification. One of the raters, however, noticed a "surface", or "format" property of the summaries and classified the summaries on this basis. This lead to one classification at a chance level and one perfect classification. Subjects in the written protocol condition tended to express their summaries in a text fashion, while subjects in the verbal condition tended to use a pseudo-programming language description. Given the nearly equal content of these summaries, we are not certain what implications, if any, this result has.

We also asked subjects to summarize the techniques and processes used to perform this task. Again, we were interested in determining how much information could be obtained by this procedure and determining whether there were any differences due to experimental condition.

Nine of the ten subjects described some form of top-down or level-structured technique. Several subjects used the term "top-down", some made statements such as "I start with the really big parts and

don't worry about the small problems till later" or "I begin with identifying a problem and breaking out specific functions that need to be done -- the WHAT rather than the HOW -- Procrastination Programming." The single exception to this type of description cited only "intuitive reasoning." The majority of subjects, therefore, claim to use a top-down mode of expansion and appear to be familiar with the concept of levels of detail within a design. An examination of their designs, however, indicates that they do not use these techniques either consistently or successfully.

In general, and especially in comparison with the designs produced by our expert subjects, none of the designs produced by this group of subjects can be considered complete. That is, although most of these subjects correctly identified the principal components of a complete design, they did not completely specify all of the necessary functions at the lower levels of the design. For example, while our expert subjects would consider such functions as "build list" or "match words" to be very complex, less experienced subects would use such functions as primitives and would not describe the actual operations or details involved.

Because of the nature of the problem involved, we would not expect these subjects' knowledge about page-keyed indexes to differ significantly from that of the expert subjects. It is reasonable, however, to expect that there would be differences between the software design knowledge of the two groups.

Seven of these subjects produced designs that can be characterized as top-down and three attempted to write programs rather than designs. In general, our less experienced subjects appear to do a very "quick

and dirty" top-down expansion of the design. Although these subjects have done quite a bit of programming, they have had little, if any, experience in designing systems that others would implement. They are used to writing "quick and dirty" programs, running the programs to locate errors, and then correcting the errors. In many respects, this is similar to Sussman's (1973) HACKER system. While our expert subjects had the requisite experience to do fairly clean expansions, these subjects are used to generating quick and dirty designs that capture the principal components of the problem but not necessarily the interactions between of details of these components, and then doing extensive modifications at the code, or operation sequence, level.

We interpret this result to mean that these subjects, through a lack of experience, have not developed a sufficient set of critics, debugging heuristics, etc., to "debug" plans or designs. In addition, the failure to expand certain critical elements of the designs suggests that these subjects, although they are well aware of the general capabilities of computers, do not fully comprehend the difficulty involved in certain operations or functions. Unlike the expert subjects, these subjects do not have fairly well developed schemata for given classes of design problems. As a result, they are unable to completely construct a solution plan without obtaining feedback from the attempted execution or implementation. This is very similar to the operation of Sussman's (1977) PSBDARP (Problem Solving By Debugging Almost-Right Plans).

It appears that the generation of plans is a schema-driven activity. If a well-developed schema exists, as in the case of our expert subjects, a plan can be developed. Some elements of the final

plan may be stored directly in a schema, and other plans may be constructed by modifying a more generic plan. Our less experienced subjects had schemata that covered only the higher levels of a plan. Their lack of experience did not support schemata sufficient to develop intermediate-level plans. In this case, they must rely on the feedback obtained from attempting plan execution and modifying the plan, at some level, to eliminate errors. In effect, when plans can no longer be successfully developed, subjects resort to some type of depth-first, failure-driven search for a solution.

A major purpose of this experiment was to examine alternative methods for the collection of experimental protocols. In this regard, this experiment was only partially successful. There were, in general, no apparent differences in the written design solutions, design summaries, or technique summaries obtained from subjects in either experimental condition. In addition, the comments of subjects in the verbal protocol condition did not appear to provide significantly more information than was contained in the written descriptions. Although these results are encouraging, the overall quality of the data obtained was much lower than that obtained from the more experienced subjects. Before concluding that equally useful information is obtained in both written protocol and verbal protocol conditions, a replication with more experienced subjects is required. The results of the current experiment are, however, sufficiently encouraging to warrant such additional studies.

CONCLUSIONS

The goal of the research described in this paper is to investigate the problem solving processes used in complex tasks and to model the manner in which these processes allow for planning behavior. Clearly, this is an ambitious goal, and the research described in this paper is only a first step toward this end. Intuitively, planning does occur in complex tasks; although the effects of planning can be observed in controlled situations, the types of plans that could be used are difficult to taxonomize, and the cognitive processes underlying planning behavior are not readily apparent. As first steps, therefore, we attempted to select an appropriate task domain, to develop a theoretical framework that could usefully guide our explorations, and to devise experimental tasks that would permit us to observe and ultimately understand planning behavior.

The experimental task that was selected was software design. We selected problems whose solution did not require extensive knowledge of computer science techniques; that is, problems that could be solved by a wide range of subjects. We selected the domain of software design because we felt that planning behavior is likely to occur in these types of problems. Any type of design is, essentially, a plan or subgoal structure that describes how some complex task is to be accomplished. In software design, people are accustomed to documenting this subgoal structure and, across designers, a fairly common terminology is used. The combination of a problem in which planning is required with a task environment in which the documentation of plans is typical seems very appropriate for investigations of planning behavior.

The starting point for the development of a theoretical framework was a fairly straightforward translation of the concept of planning by

abstraction, as discussed by Newell and Simon (1972), and further employed, in the domain of artificial intelligence, in Sacerdoti's (1975) concept of a procedural net. Although a procedural net corresponds most closely to what we refer to as detailed plans, we assumed that detailed plans were derived from abstract plans and further refined into the sequence of operations that actually implement a solution to a problem.

There are three aspects of Sacerdoti's NOAH that are particularly relevant to investigations of planning. These are: the final form of the plan, the dynamics of plan construction, and the knowledge structures that guide planning. In the paragraphs below, we will discuss each of these aspects as they relate to planning in the context of software design.

First, our theoretical framework predicts that the final form of a software design can be characterized as a procedural net. As we indicate in Appendix A, which reviews formal software design practices, there is general agreement among computer scientists that software designs take the form of a hierarchical structure. In addition, we were able to represent the final designs of all of our expert subjects in this manner. In its ability to characterize completed plans, we suggest that the concept of a procedural net is both useful and general.

The second aspect of a procedural net concerns the manner in which plans are generated. As described by Sacerdoti (1975), procedural nets must be expanded in a top-down, breadth-first manner. This is a very strong claim and, as evidenced by our experimental results, is not, in general, true. To be sure, some of the planning behavior that we observed could be characterized in this manner. Other subjects,

however, constructed plans in very different fashions.

The third aspect concerns the organization of the knowledge that guides planning. In NOAH, it is assumed that the knowledge necessary to successfully decompose a given node is directly stored in that node. This does not facilitate consideration of what is common among these knowledge sources and how they are organized. For this reason, we did not incorporate the NOAH knowledge organization structure into our initial theoretical framework. Rather, we attempted to extend this concept to consider the types of knowledge that are involved in the synthesis of a procedural net. We assume that these knowledge structures are essentially families of hierarchical schemata. As such, they represent well-learned techniques, patterns etc. In view of this, we characterize planning as a process of synthesis which, in large part, involves the novel combination of well-known elements.

As a result of this activity, we concluded that there is a great deal of similarity between our framework and the HEARSAY-like model of problem solving proposed by Hayes-Roth and Hayes-Roth (1978). Both assume that behavior is determined by the synthesis of diverse knowledge structures. Originally, we identified two principal knowledge structures -- one concerned with understanding and solving the problem as given and the other concerned with developing a software design to achieve a solution. A HEARSAY-like framework appears to provide a convenient formalism for considering more detailed aspects of these and other knowledge structures, and we intend to pursue this issue in future research.

A HEARSAY-like formalism may also be useful for resolving the second aspect of a procedural net discussed above, i.e., the dynamics of plan construction. Clearly, a plan can be generated in a large

number of ways. By postulating a number of knowledge structures, it may be possible to describe different modes of expansion as different selections from a common set of knowledge structures.

We also conclude, however, that a great deal of planning behavior involves the retrieval and modification of previously internalized schemata. In this regard, we feel that Sussman's (1977) PSBDARP (Problem Solving By Debugging Almost-Right Plans) is particularly relevant. As with the HEARSAY-like models, we feel that this type of model is compatible with, and could usefully extend, our theoretical framework.

The principal conclusion that we derive from our present studies is that planning behavior is schema-driven. Plans are derived from a subject's previous experiences in similar problems. Useable plans may be directly accessed from an existing schema, an existing plan may be modified to better fit the current situation, or a plan, or subgoal structure, may be constructed "from scratch." The protocols of our three expert subjects show fairly clear differences in the overall quality, completeness, or organization of these previously developed schemata.

In interpreting the diverse types of behavior observed in our expert subjects, we began with two assumptions. The first is that all three subjects share a fairly common knowledge of the task environment of software design. That is, we assume that there were not significant differences in these subjects' expertise in this area. Second, we assume that there were no differences in these subjects' knowledge of indexes for textbooks, which was the specific problem used.

We concluded, therefore, that the only substantive difference in these subjects' behavior is due to general knowledge structures that

control the synthesis of the above types of knowledge. In particular, we concluded that these knowledge structures are previously developed, generic problem schemata that indicate how knowledge of software design and indexers can and should be merged in order to produce an appropriate design. In one of our subjects (S2), we feel that these knowledge structures were sufficiently developed that they essentially drove a top-down expansion. In the second subject (S3), these knowledge structures were less well developed and, as a result, backtracking was necessary in order to correct observed deficiencies in the design. In the remaining expert subject (S5), we contend that these knowledge structures were developed to such an extent that this subject was able to retrieve a memory structure that essentially described the entire design. In this case, this subject was then able to isolate and expand the single module that most affected the efficiency of the implementation, which, for this subject, was an important aspect of a software design.

In our less experienced subjects, the appropriate schemata were not completely developed. Although these subjects had schemata for the higher levels of a plan, they lacked appropriate schemata for developing more detailed plans. At this point, we argue, these subjects ceased to plan and resorted to some type of depth-first, failure-driven search for a solution. This type of behavior is common in transformation problems, where it is unlikely that planning can occur with naive subjects, and the search process is guided by a means-ends heuristic. With the experimental problem used in these studies, our less experienced subjects could identify some components of the goal state and were attempting, at a very low level of detail, to manipulate these components.

In summary, we have described an initial investigation of the problem solving processes used in complex tasks. A possible criticism of our current work is that we are concentrating on memory representation issues and not directly on problem solving, or knowledge utilization, processes. While we do not accept a strict demarcation between the areas of knowledge representation and knowledge utilization, there is some truth to this criticism. In previous work on problem solving (e.g., transformation problems), the effects of knowledge representation issues have been minimal. Similarly, the effects of knowledge utilization have little impact on many of the studies of knowledge representation, such as the comprehension of short stories. In more complex tasks, however, which Bhaskar and Simon (1977) refer to as "semantically rich", questions of knowledge representation and knowledge utilization are highly interrelated.

We have characterized planning as a process of synthesis, or as the "novel combination of well-known elements." Our current work, therefore, focuses both on identifying these "well-known" elements and the processes that guide their "combination."

On the methodological side, we have explored various procedures for collecting protocol data. Complete protocols provide very rich, extensive sources of information. In complex tasks, however, the extensiveness of this form of data makes an objective analysis extremely difficult. The development of empirical techniques to simplify data collection and analysis, without reducing the quality of the data that is obtained, is a desirable, if not necessary, component of investigations of behavior in semantically rich domains.

APPENDIX A

SOFTWARE DESIGN PRACTICES

Within the last several years, several prescriptive technqiues for software design have been proposed. In this section, we will review some of the more prominent software design practices. This section is not intended to be an exhaustive, in-depth review, but rather is intended to provide the reader unfamiliar with this area with a general overview of currently used design techniques and the concepts involved in these techniques. Further, we will limit our consideration to software design techniques, per se, and exclude other software development practices such as "structured walkthroughs" and "egoless programming", with which the reader may be familiar.

In our discussion of task domains, we characterized software design as the process of translating functional specifications into a structural description of a computer system that would satisfy these specifications. A common assumption among all software design techniques is that this description takes the form of a hierarchical structure. The principal difference among the various techniques concerns the manner in which this structure is expanded or generated.

We use the term "software design technique" to refer to fairly formal rules or guidelines for performing a software design. We emphasize rules or guidelines to differentiate between these techniques and the much less formal "techniques", such as "bottom-up" and top-down" with which the reader may be more familiar. In this section, we will first consider these less formal approaches and then consider more procedural techniques.

A review of some of these less proceduralized techniques is

presented by Boehm (1975), who also considers the relative advantages and disadvantages of thse techniques. The techniques considered by Boehm are "bottom-up, two variations of "top-down", structured programming", and "model-driven."

When using a bottom-up approach, a designer must first identify those functions or routines whose development seems most "important" to the overall design. "Importance" can be defined in terms of efficiency, cost, development effort, etc. As the term "bottom-up" implies, these functions are at the lower levels of the hierarchical structure that is being developed to represent the design. Once these routines are developed, the designer develops a "test driver" to allow testing of these modules and their interactions, a "computation monitor" to control the order in which these functions are executed, and any necessary input-output modules. Finally, input-output "controllers", initialization routines, and similar procedures, are developed and the entire design is then tested for errors.

In general, a bottom-up approach involves constructing low level routines and then constructing "drivers" to control interactions between the low level routines. There are two primary advantages to this approach. First, "high risk" components (e.g., processing natural language, real time sensors, etc.) can be identified early. If it is determined that it is not feasible to implement these components as originally specified, the design specifications can be changed before a great deal of effort is expended. Second, the emphasis on the lower levels encourages the development of reuseable modules that can be applied to other designs with little or no modification.

A primary disadvantage of this approach is that very little

attention is given, early in the design process, to the interactions between modules. It may well be the case that interactions between modules present more problems than the development of the individual modules. In addition, a bottom-up approach does not give a great deal of attention to overall system requirements, including user interfaces and data structures and, in an effort to use the lower level components that are already developed, the higher levels of the design may be "patched up". As a result, the total design may be very difficult to implement, understand, or modify.

The type of "top-down" design with which most readers will be familiar is termed by Boehm as the "top-down stub" approach. In this approach, the designer first considers the overall system requirements and develops a top level program to meet these requirements. This top level contains the necessary logic to control the lower level functions which are initially represented as "stubs". In successive design steps, these stubs are then decomposed into control logic and necessary subfunctions, which are also represented as stubs.

The advantages of a bottom-up approach, identification of high risk components and development of reuseable modules, are disadvantages of a top-down stub approach. In contrast, the advantages of this approach are early attention to the interactions between modules and a more coherently defined higher level in the design, which allows for easier testing and maintainability. Neither of these two approaches, however, explicitly considers the possibility that the user's requirements are not properly stated in the original problem statement, although obvious discrepancies could be detected earlier and with less effort with a top-down approach.

There are several variations of a "top-down problem statement" approach, each restricted to a specific problem domain. This restriction is due to the fact that such approaches generally involve specialized languages for expressing the design requirements and the final design. One example of this approach is ISDOS (e.g., Teichroew and Sayani, 1971) and its Problem Statement Language. Other examples, such as ISDS (Hamilton and Zeldin, 1976) also employ specialized languages that, through the allowed constructs, enforce the use of what are considered "good" design practices in a particular domain. For example, the language may allow only certain types of interfaces between modules.

This approach has all of the advantages of the top-down stub approach plus more explicit consideration of user requirements. The primary disadvantage is the limited application areas of any version of this approach.

We have mentioned the top-down problem statement approach primarily for the sake of completeness. This approach provides a large number of aids to the designer and restricts the types of decisions that the designer can make. Since we are interested in observing planning behavior in the context of software design, these restrictions make this approach unsuitable for our present purposes. For investigating the effects of various types of problem solving aids, however, this may be an appropriate area of study.

The "structured programming" approach to design is a direct extension of structured programming concepts (e.g., Dahl, Dijkstra, and Hoare, 1972) to the design process. The principal concepts are the use of hierarchical, modular structures, the use of only "structured"

control structures, and the requirement that each module have a single input and output. This approach is compatible with the other approaches mentioned in this section and is especially useful when demonstrations of design "correctness" are important.

"Model-driven design" attempts to relate the "requirements that are to be satisfied with the "properties" of the computer system involved, frequently through a matrix representation. Design generally proceeds in a top-down fashion, but the use of such a matrix allows the early identification of high risk components that may be best developed in a bottom-up fashion. This technique has not been extensively used and appears to describe the management of design activities more than the actual processes involved in design.

Other fairly general design techniques have also been mentioned in the software design literature. "Middle-out" design requires the designer to identify and initially develop the most "important" routine or function; in this regard, this approach is similar to a "bottom-up" approach. The primary difference is that this routine need not be the the lowest level of the final design. Rather than being function oriented, as in bottom-up design, the identified routine could be control-oriented, input-oriented, etc. In general, this routine is selected because of constraints on the final implementation, such as hardware constraints, user interface considerations, etc.

Like a bottom-up approach, designing middle-out tends to lead to the early identification and development of high risk components. The principal disadvantage is that the remainder of the design may be "patched up" to work with the first routine developed so that this high risk component will not have to modified. Also like a bottom-up

approach, this technique may involve the modification and use of previously developed modules. The actual advantages and disadvantages of this approach depend on where in the final design structure the initially developed module falls, since a middle-out approach could, conceivably, proceed in a strictly top-down or bottom-up fashion.

With interactive systems, design may proceed in either an "inside-out" or "outside-in" manner. An inside-out approach begins with a description of basic implementation environment abilities and functions and attempts, through adding higher level modules, to match these basic abilities and functions to user requirements. An outside-in approach, on the other hand, begins with a description of the user requirements and attempts to work down toward the available capabilities. While an inside-out approach leads to the development of a very efficient design in terms of hardware and software, an outside-in approach tends to insure that the initial statement of user requirements is practical, and, if this is not the case, leads to an early reformulation of these requirements.

One purpose of this review is to introduce some of the design practices and techniques that subjects in our experiments might employ. In this regard, model-driven and top-down problem statement approaches are not of particular interest and we have included them primarily for the sake of completeness. The emphasis of these two approaches, on management and limited application domains, as well as the use of specialized languages and design aids suggests that these approaches would not be useful in characterizing subjects' behavior in the type of design task involved in the research reported in this paper. In addition, we do not consider the "structured programming" approach to

design to be unique, since the emphasis is on constraints that the design must satisfy and not on how the design should be expanded. In general these constraints could be applied to any of the other approaches described above.

The remaining approaches bottom-up, top-down stub (which we will refer to as "top-down"), middle-out, inside-out, and outside-in should be useful in characterizing subjects' behavior. We would not expect, however, that a given subject would always select one of these approaches and ignore the others. For example, if a subject has had a great deal of experience in a given application domain, and has developed several possibly relevant modules, this may lead to the selection of a bottom-up or middle-out approach so that these previously developed modules can be reused. On the other hand, if the designer expects that the interactions between modules may be a crucial issue, a top-down approach could be selected. In general, the approach that is selected is influenced by the designer's previous experiences and the nature of the current design problem.

These approaches describe how design should be done at a very general level. They do not specify the actual steps involved in constructing a design in any formal of procedural way. In addition, they do not provide explicit criteria along which the final design or the current state of a developing design can be evaluated.

Within the past few years, however, software design techniques have been proposed that provide both proceduralized descriptions of how the design process should proceed and criteria for evaluating designs. These techniques dictate to the designer, in some detail, how a design should be expanded. Consequently, what we may, at first glance,

interpret as planning behavior on the part of the designer may actually be the sequential application of a well learned set of procedures or rules. In effect, these techniques represent schemata for how designs should be accomplished, regardless of the particular application domain involved.

One of these techniques is "Structured Design." The initial emphasis of "Structured Design", as advocated by Stevens, Myers, and Constantine (1974), Myers (1975) and Yourdon and Constantine (1975) is on the flow of data through the system being designed. The design problem is first restated in terms of a "data flow" or "bubble chart" that identifies the major functions involved in transforming the "input" data into the appropriate "output" data.

Following this step, afferent (incoming) and efferent (outgoing) data flow boundaries are identified. When data first enters the system it is classified as "input". After several processing steps, however, it becomes more "abstract" and is less easily characterized as input. Similary, "output" data, at some earlier point, is less easily charcterized as "output". Using the data flow chart, the "point of highest abstraction" can be identified for both input and output data. The function that connects these two points is called the "central transform."

At this point, a design decision is made. If the data flow graph produces a central transform that splits an input stream into several discrete output streams (an "OR" relationship), then "transaction analysis" is the appropriate design technique; if the structure is essentially linear (and "AND" relationship), then "transform analysis: is called for.

In transform analysis, each function in the data flow chart is identified as a "source", "sink", or "transformer" of data. Next, the functions of all identified modules and their interfaces are described. Level by level and breadth-first, each module is expanded into its subfunctions by applying the following steps: each identified transform can be treated as a "central transform", "source" modules may be decomposed into transforms and afferent modules, until the ultimate physical input is reached, and "sink" modules are decomposed into transforms and efferent modules.

In transaction analysis, design centers on the "transaction center." First, for each transaction, a transaction module is created to perform the indicated process. Each transaction module is further decomposed into "action" modules that indicate the actions involved in the transaction, and each action module is further decomposed into "detail" modules, which indicate the detailed steps involved in the various actions.

Regardless of which type of analysis is used, the principal criteria along which designs are evaluated are "coupling" and "cohesion." Coupling is a measure of the interdependence between modules. Coupling is to be minimized (independence is to be maximized), and certain types of coupling are more desirable than others. For example, shared data should be "local" to the coupled modules rather than "common" or "global". Cohesion is the degree to which a given module performs a single, well-specified objective, and high cohesion is desired.

The approach advocated by Jackson (1975, 1977) begins by considering the structure, rather than the flow of data. This approach

is much more "algorithmic" than structured design in that there are no decision points encountered in the design process. Jackson assumes that the structure of the design must correspond, in a one-to-one fashion, with the structure of the data.

Both the input and output data structures are represented in terms of three constructs -- sequence, iteration, and selection. These constructs, that are derived from the concept of "structured programming" are considered adequate to represent both data structures and designs. If the structures of both the input and output data correspond exactly, there is no problem; if they do not correspond, then a "structure clash" exists. This structure clash is resolved by "program inversion", which may involve the use of intermediate data files, coroutines, or reversing the calling order between modules.

Once the data structures are defined, the design structure is created from them. The purpose of a design is to produce a mapping between the input data and output data. The design is created by including a selection, sequence, or iteration component for each corresponding component in the data structures.

Warnier's (1974) "Logical Construction of Programs" (LCP) approach is very similar to that of Jackson. Warnier's approach, however, is somewhat more algorithmic. The first step in LCP is to reformulate the data structure, producing, in most cases, a hierarchy. Each decomposition point in this hierarchy is represented either as "repeated information" or "alternate possibilities" (similar to Jackson's iteration and selection), and data items within each level of the hierarchy are presented sequentially. On the basis of this structure, a skeleton program structure, outlining the flow of

execution, is created.

When the skeleton structure is completed, the operations or actions that are required by each part are listed. These functions are then sorted on the basis of the function they perform (input, output, calculation, etc.) and assigned to the proper part of the skeleton. Following, this, the skeleton structure, which now represents the design, is verified by comparing it with the analyzed data structures.

The three design techniques outlined immediately above present fairly formal and proceduralized approaches to performing a design task. As the reader may have noticed, there are several similarities between the techniques involved in structured design and the theoretical framework that we present in the main body of this report. For example, both indicate that designs are expanded in a top-down, breadth-first manner. In addition, what we identified as a very general "input-process-output" schema is evidenced in structured design's emphasis on afferent, efferent, and transform modules, and the decomposition of "action" modules into "detail" modules is consistent with our assumption that there are multiple levels of detail in a plan or procedural net.

An additional software design procedure, termed "stepwise refinement" by Wirth (1971), also deserves consideration. There are several striking similarities between this approach and the theoretical framework presented in this report. This approach is less procedural than the formal design techniques described immediately above, but provides more detail about how a design should be expanded and evaluated than the less formal approaches discussed earlier in this section. Several variations of this approach will be considered

briefly.

In stepwise refinement, the designer starts at the top level of the design, which is essentially a statement of the goal "solve the problem". Design then proceeds in a breadth-first, level-by-level manner. These levels can be differentiated with respect to the amount of detail involved. At the early levels, the designer does not consider specific programming languages or other aspects of the environment in which the solution will be implemented. As Ledgard (1973, pp. 45-46) points out, this stage of the design might contain statements like "compute the nth prime number", "find the roots of the equation", or "process the payroll". We would characterize this as the abstract plan level. Toward the lower levels of the design, the operation sequence level, the design works in terms of the implementation environment. The intermediate levels of the design, the detailed plan level, represent a transition between these very general and very specific expressions.

As just described, the process of stepwise refinement is similar to the processes that we assume underlie the expansion of a procedural net. Although one approach is based on theoretical assumptions and the other on the pragmatic considerations involved in developing reliable software, the underlying concepts are quite similar. Although, several variations of stepwise refinement have been proposed, they all adhere to these basic concepts.

The main contribution made to design methodologies by Parnas is associated with the phrase "information hiding" (Parnas, 1972). Parnas argues that the primary criterion for a modular decomposition is that every module can be characterized by its "knowledge of a design

technique, which it hides from all others." In general, Parnas' technique is similar to the usual procedures of stepwise refinement outlined above.

Ledgard (1973) extends the definition of stepwise refinement by incorporating Mill's general top-down concepts (e.g., Mills, 1971) and Dijkstra's definition of structured programming. This technique, called "meta-stepwise refinement" by Peters and Tripp (1977) provides a clear expression of the general concepts underlying stepwise refinement. Ledgard's approach has six primary characteristics (pp. 46-47). First, the designer must develop a clear understanding of the problem before proceeding. Second, the initial stages of the design are independent of considerations of the implementation environment; such considerations are only included at lower levels. Third, design is done in discrete levels, although Ledgard admits the possibility that it may be useful to "look ahead" to the probable functions of a lower level. That is, some design decisions may be based on the practicality or risk of the ultimate functions or modules that may be required by these decisions. Fourth, "the programmer concentrates on critical, broad issues at the initial levels, and postpones details until lower levels." Fifth, the designer must ensure that each level represents, at the appropriate level of detail, a correct solution to the problem. Finally, each level is generated by "successive refinement" of the preceding level.

Liskov (1972) presents criteria for selecting an appropriate "level of abstraction" for each level in a design. According to Liskov, the first step in a design is to reformulate the problem into a set of abstractions that are necessary and sufficient to satisfy the

system constraints and then to make these abstractions more concrete, in the sense of expressing them in terms of implementable functions. Liskov identifies several types of abstractions that are to be used in deciding on appropriate modular decompositions.

"Goal directed programming", as advocated by Cichelli and Cichelli (1977) also extends the concept of stepwise refinement. The addition made by this technique is the concept of an explicit statement of the goal to be achieved by the design. This approach involves stating the goal to be achieved, deriving an assertion that will be affirmed when the goal is true, and then deriving a logical condition from this assertion that will become true when the assertion becomes true. By iterating these steps, the original goal is decomposed into subgoals. At a general level, this technique is similar to the use of a means-ends analysis heuristic. Like structured design, the principal criteria along which designs are evaluated are cohesion and coherence. The emphasis on using only iteration, selection, and sequence constructs and the stipulation that design structure should correspond to data structure is similar to the approaches of Jackson and Warnier.

Although this review is brief, it is intended to provide the reader with some feel for the diversity of techniques that could be applied in a software design task. In addition, this review may aid the reader in interpreting some of the software designs that were produced in the experiments reported in this paper. Structured design and the techniques advocated by Jackson and Warnier are, to a designer who rigorously follows one of these approaches, schemata for how a design task should be performed. As such, what could be considered to be planning behavior on the part of the designer may reflect the

successful application of the procedures called for by these approaches.

Top-down, bottom-up, and the related very general design approaches are essentially strategies for the construction of plans, in the context of design. At a very general level, they indicate the types of plans that are allowable or that may be useful. They do not, however, indicate how these plans are to be constructed.

The remaining approaches considered in this section, which are all variants of a stepwise refinement approach, are somewhat more explicit about the criteria that should be used to evaluate plans, or designs, than are the more general approaches. Although they give guidelines for expanding a design, they do not prescribe formal procedures.

REFERENCES

Atwood, M. E., & Polson, P. G. A process model for water jug problems. Cognitive Psychology, 1976, 8, 191-216.

Banerji, R. B., & Ernst, G. W. A comparison of three problem solving methods. Proceedings of the International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, 1977, 442-449.

Bhaskar, R., & Simon, H. A. Problem solving in semantically rich domains: An example from engineering thermodynamics. Cognitive Science, 1977, 1, 193-215.

Boehm, B. W. Software design and structuring. In E. Horowitz (Ed.), Practical strategies for developing large software systems. Reading, Massachusetts: Addison-Wesley, 1975, 104-128.

Chase, W. G., & Simon, H. A. Perception in chess. Cognitive Psychology, 1973, 4, 55-81.

Cichelli, R. J., & Cichelli, M. J. Goal directed programming, SIGPLAN Notices, 1977 (July), 12(7), 51-59.

Dahl, O. J., Dijkstra, E. W., & Hoare, C. A. R. Structured programming. New York: Academic Press, 1972.

de Groot, A. D. Perception and memory versus thought: Some old ideas and recent findings. In B. Kleinmuntz (Ed.), Problem solving: Research, method, and theory. New York: Wiley, 1966.

Duncker, K. On problem solving. Psychological monographs, 1945, 58, No. 5 (Whole No. 270).

Egan, D. E. The structure of experience acquired while learning to solve a class of problems (Unpublished doctoral dissertation) Ann Arbor, Michigan: University of Michigan, Department of Psychology, 1973.

Egan, D. E., & Greeno, J. G. Theory of rule induction: Knowledge acquired in concept learning and problem solving. In L. Gregg (Ed.), Knowledge and cognition. Hillsdale, New Jersey: Erlbaum, 1974.

Greeno, J. G. The structure of memory and the process of solving problems. In R. L. Solso (Ed.), Contemporary issues in cognitive psychology. Washington, D. C.: Winston, 1973.

Greeno, J. G. Hobbits and orcs: Acquisition of a sequential concept Cognitive Psychology, 1974, 6, 270-292.

Greeno, J. G. Process of understanding in problem solving. In N .J. Castellan, D. B. Pisoni and G. R. Potts (Eds.), Cognitive theory, Vol. 2. Hillsdale, New Jersey: Erlbaum, 1977, 43-82.

Greeno, J. G. Natures of problem solving abilities. In W. K. Estes (Ed.), Handbook of learning and cognitive processes, Vol. 5. Hillsdale, New Jersey: Erlbaum, 1978, 239-270.

Hayes-Roth, B., & Hayes-Roth, F. Cognitive processes in planning (Technical Report No. RAND/WN-10268-ONR). Santa Monica, California: Rand Corporation, August 1978.

Hinsley, D. A., Hayes, J. R., & Simon, H. A. From words to equations: Meaning and representation in algebra word problems (CIP Working Paper No. 331). Pittsburgh, Pennsylvania: Carnegie-Mellon University, October 1976.

Hamilton, M., & Zeldin, S. Integrated software development system/higher order software conceptual description (Technical Report ECOM 76 0329 F). Fort Monmouth, New Jersey: U. S. Army Electronics Command, 1976.

Jackson, M. A. Principles of program design. New York: Academic Press, 1975.

Jackson, M. A. The Jackson design methodology. In P. Freeman & A. I. Wasserman (Eds.), Tutorial on software design techniques. Long Beach, California: IEEE Computer Society, 1977, 219-234.

Jeffries, R., Polson, P. G., Razran, L., & Atwood, M. E. A process model for missionaries-cannibals and other river crossing problems. Cognitive Psychology, 1977, 9, 412-440.

Kintsch, W., & van Dijk, T. A. Comment on se rapelle et on resume des histories. Langages, 1975, 40, 98-116.

Kintsch, W., & van Dijk, T. A. Toward a model of text comprehension and production. Psychological Review, 1978, 85, 363-394.

Knuth, D. The art of programming: Vol. 1. Fundamental algorithms. Reading, Massachusetts: Addison-Wesley, 1968.

Larkin, J. H. Problem solving in physics (Technical Report). Berkeley, California: University of California, Department of Physics, July, 1977.

Ledgard, H. F. The case for structured programming. Bit, 1973, 13, 45-47.

Levin, S. L. Problem selection in software design (Technical Report No. 93). Irvine, California: Department of Information and Computer Science, University of California, November 1976.

Liskov, B. H. A design methodology for reliable software systems. AFIPS Conference Proceedings, 1972, 41 (Part 1), 191-199. Reprinted in P. Freeman & A. I. Wasserman (Eds.) Tutorial on software design techniques. Long Beach, California: IEEE Computer Society, 1977, 53-61.

Long, W. J. A program writer (Technical Report No. MIT/LCS/TR-187). Cambridge, Massachusetts: Massachusetts Institute of Technology, Laboratory for Computer Science, November 1977.

Miller, G. A., Galanter, E., & Pribram, D. H. Plans and the structure of behavior. New York: Holt, Rinehart, and Winston, 1960.

Mills, H. D. Top-down programming in large systems. In R. Rustin (Ed.), Debugging techniques in large systems. Englewood Cliffs, New Jersey: Prentice Hall, 1971.

Minsky, M. A framework for representing knowledge. In P. H. Winston (Ed.) The psychology of computer vision. New York: McGraw Hill, 1975.

Myers, G.J. Software reliability: Principles and practices. New York: Wiley, 1975.

Newell, A., Shaw, J. C., & Simon, H. A. Chess playing programs and the problem of complexity. IBM Journal of Research and Development, 1958, 320-335. (Reprinted in E. A. Feigenbaum & J. Feldman (Eds.), Computers and thought. New York: Mc-Graw-Hill, 1963.)

Newell, A., & Simon H. A. Human Problem Solving. Englewood Cliffs, New Jersey: Prentice Hall, 1972.

Peters, L. J., & Tripp, L. L. Comparing software design methodologies. Datamation, November 1977, 89-94.

Parnas, D. L. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 1972, 15, 1053-1058.

Sacerdoti, E. D. Planning in a hierarchy of abstraction spaces. Artificial Intelligence, 1974, 5, 115-135.

Sacerdoti, E. D. A structure for plans and behavior. (Technical Note 109). Menlo Park, California: Stanford Research Institute, August 1975.

Schank, R. C. Conceptual dependency: A theory of natural language understanding. Cognitive Psychology, 1972, 4, 552-631.

Schank, R. C., & Abelson, R. P. Scripts, plans, goals and understanding: An inquiry into human knolwedge structures. Hillsdale, New Jersey: Erlbaum, 1977.

Simon, H. A., & Reed, S. K. Modeling strategy shifts in a problem solving task. Cognitive Psychology, 1976, 8, 86-97.

Stevens, W. P., Myers, G. J. & Constantine, L. L. Structured design. IBM Systems Journal, 1974, 13, 115-139.

Sussman, G. J. A computational model of skill acquisition (Technical Report No. 297). Cambridge, Massachusetts: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1973.

Sussman, G. J. Electrical design: A problem for artificial intelligence research. Proceedings of the International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, 1977, 894-900.

Teichroew, D., & Sayani, H. Automation of system building. Datamation, 1971, August, 25-30.

Thomas, J. C., Jr. An analysis of behavior in the hobbits-orcs problem. Cognitive Psychology, 1974, 6, 257-269.

Warnier, J. D. Logical construction of programs. Leiden, Netherlands: Stenpert Kroese, 1974.

Wasserman, A. I. Case studies in software design. In P. Freeman & A. I. Wasserman (Eds.), Tutorial on software design techniques. Long Beach, California: IEEE Computer Society, 1977.

Wertheimer, M. Productive thinking. New York: Harper-Row, 1945. (Revised 1959).

Winograd, T. A program for understanding natural language. Cognitive Psychology, 1972, 3, 1-191.

Wirth, N. Program development by stepwise refinement. Communications of the ACM, 1971, 14, 221-227.

Yourdon, E. ,& Constantine, L. L. Structured design. New York: Yourdon, Inc., 1975.